1. Jbs1000-Getting Started
2. Jbs1010-Objects and Encapsulation
3. Jbs1020-Classes
4. Jbs1050-A Roadmap
5. Jbs2000-What is Sound?
6. Jbs2010-Your First Sound Program
7. Jbs2020-Square Wave Sound
8. Jbs2030-A Pure Sinusoidal Tone
9. Jbs2040-An Audio Graph of a Sinusoid
10. Jbs2050-Runtime Polymorphism with Java Sound
11. Jbs2060-A Player Piano Simulator
12. Jbs2070-A General Purpose AudioGraph Program

Jbs1000-Getting Started
This module explains how to get started programming in Java in a format that is accessible to blind students.

Table of Contents

Preface

This module is one in a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java with particular emphasis on accessibility for blind students.

This module, along with several modules that follow provide the background information that you will need to understand the sound and audio programming material later in the course. You will need to study this material if you are new to Java programming.

On the other hand, if you already have Java programming experience, or if you are just curious, you may want to jump ahead to the module titled *Jbs2000-What is Sound?*

Prerequisites

As mentioned in an earlier module, in addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in this and the following modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ .
- A refreshable Braille display capable of providing line by line tactile output of information displayed on the computer monitor is recommended. Such a display is described at http://www.userite.com/ecampus/lesson1/tools.php , is recommended.
- The Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/8/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in this and the following modules include:

- A cursory understanding of algebra.
- An understanding of the material covered in the *Programming Fundamentals* modules that you will find in two formats at the following URLs. These modules provide fundamental programming concepts using the Java programming language in a format that should be accessible.
    - http://cnx.org/content/m45179/latest/?collection=col11441/latest
    - http://cnx.org/contents/fb64661c-5b3f-4ea8-97c6-e48df112438a

The essence of OOP

My dictionary provides several definitions for the word essence. Among those definitions are the following:

- The property necessary to the nature of a thing
- The most significant property of a thing

Thus, this and several modules that follow describe and discuss the necessary and most significant aspects of OOP using Java -- the essence of OOP using Java. For the first few modules, I will provide that information in a high-level format, devoid of any requirement to understand detailed Java syntax. In those cases where an understanding of Java syntax is required, I will provide the necessary syntax information in the form of supplementary notes.

If you understand the fundamentals of computer programming described above , you should be able to read and understand the modules in this miniseries.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Preview

Three important concepts

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module will concentrate on encapsulation. Encapsulation will be used as a springboard for a discussion of objects.

A description of an object-oriented program will be provided, along with a description of an object, and how it relates to encapsulation.

A car radio

I suspect that many of you grew up in a family where the family car contained a push-button radio and I also suspect that you learned how to operate that radio. In fact, as children, many of you may have quarreled with your siblings about which button to push and which radio station to listen to.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects.

What you will learn

For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects. You will learn that a Java object is an *instance* of a class.

You will see a little bit of Java code, used to create an object, and then to send a message to that object *(call a method on the object)* .

You will learn about Java references and reference variables. You will also learn a little about memory allocation for objects and variables in Java.

Discussion and sample code

Purpose of the miniseries

As mentioned earlier, I will describe and discuss the necessary and most significant aspects of OOP using Java.

The three pillars

Most books on OOP will tell you that in order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Begin with encapsulation

Generally, speaking, these three concepts increase in difficulty going down the list from top to bottom. Therefore, I will begin with Encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program?

Many authors would answer this question something like the following:

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object?

An object in OOP is a software construct that encapsulates data, *(along with the ability to use or modify that data)* , into a software entity.

What is encapsulation?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

"Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface.

The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. However, I will try to illustrate it in the paragraphs that follow.

A real-world analogy

Abstract concepts, *(such as the concept of an object or encapsulation)* , can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the push-button radio in a car.

The ability to store data

Most car radios have the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data*

through use of the human interface that is provided by the manufacturer of the radio.)

The data that can be stored in a car radio includes a list of five or more frequencies that correspond to your favorite radio stations.

Using the stored data

The radio provides a mechanism *(human interface)* that allows you to use the data stored therein.

When you press one of the frequency-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)*

If you have previously stored a favorite frequency in the storage location corresponding to that button, pressing the button *(sending the message)* will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency.
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

Please change your state

By following this procedure, I "send a message" to the radio object asking it to "change its state". The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. *(Alternately, we might say that the radio object sent a message back to me.)*

We say that an object has changed its state when one or more data values stored in the object have been modified.

We also say that when an object responds to a message, it will usually perform an action, change its state, return a value, or some combination of the above.

Please perform an action

Following this, when I press that button *(send a message)* , the radio object will be automatically tuned to that frequency.

Note: Historical note:
While the ability to cause your car radio to remember your list of favorite stations may seem like a miracle of modern digital electronics, the truth is that radios had this capability long before they contained digital electronics. My first car had a radio that accomplished this feat using strings, pulleys, and levers. In fact, the radio in the first car that I owned many years ago is the first programmable device of any complexity that I can remember.
As I recall, in order to set the frequency for a button, I had to manually tune the radio to a station by turning a knob, pull one of the buttons out about a quarter of an inch, and then push it in again. From that point until I did the same thing again, whenever I pressed that button, some kind of a mechanical contraption caused a big rotary capacitor to turn just the right amount to tune for a particular radio station.
Also, I remember my grandfather having a table-model radio in the early 1940's that had radio buttons. He used them to select his favorite stations, as he surfed the airwaves.

(Interestingly, the term radio button has now become a part of programming jargon, signifying certain visual components used in graphical user interfaces.)

Enough of that, now back to my modern car radio

If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set* the frequency value associated with that button to correspond to one of the radio stations in Dallas. *(Again, I would be sending a message to the radio object asking it to change its state.)*

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the *state* of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

Note: Identical objects with identical states:
It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the frequency-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action *(tune to a particular station)* . That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Calling a method

Java-speak is a little more specific than general OOP-speak. In Java-speak, we might say that pressing one of the selector buttons on the front of the radio *calls a method* on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Java is to cause that object to execute one of its methods.

Similarly, we might say that storing a new frequency that corresponds to a particular button calls a *setter* method on the radio object.

(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)

Behavior

In addition to state, objects are often also said to have *behavior* . The overall behavior of an object is determined by the combined behaviors of its individual methods.

For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers.

Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from?

In order to mass-produce car radios, someone must first create a set of plans, *(drawings, or blueprints)* for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A class definition is a set of plans

The same is true for software objects. In order to create a software object in Java, it is necessary for someone to first create a plan.

In Java, we refer to that plan as a *class* .

The class is defined by a Java programmer. Once the class definition is available, that programmer, *(or other programmers)* , can use it to produce millions of nearly identical objects.

(While millions may sound like a lot of objects, I'm confident that since Java was released into the programming world around 1997, Java programmers around the world have created millions of objects using the standard Java class named ***Button*** *.)*

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a brand new radio and say that it is an *instance* of the

plans used to produce the radio. *(Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.)*

However, it is common jargon to refer to a software object as an *instance of a class* .

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

A little bit of code

It is time to view a little bit of Java code.

Assuming that you have access to a class definition, there are several different ways that you can create an object in Java. The most common way is using syntax similar to that shown in Listing 1 below.

Listing 1 . Instantiating a new Radio object.

```
Radio myObjRef = new Radio();
```

What does this mean?

Technically, the expression on the right-hand side of the equal sign in Listing 1 applies the ***new*** operator to a *constructor* for the class named **Radio** in order to cause the new object to come into being and to occupy memory.

(Suffice it at this point to say that a constructor is code that assists in the creation of an object according to the plans contained in a class definition. The primary purpose of a constructor is to provide initial values for the new object, but the constructor is not restricted to that behavior alone.)

A reference to the object

The right-hand expression in Listing 1 returns a *reference* to the new object.

What can you do with a reference?

The reference can later be used to send messages to the new object *(call methods belonging to the new object)* .

Saving the reference

In order to use the reference later, it is necessary to save it for later use.

The expression on the left-hand side of the equal sign in Listing 1 declares a variable of the type **Radio** named **myObjRef** .

(Because this type of variable will ultimately be used to store a reference to an object, we often refer to it by the term ***reference variable*** *.)*

What does this mean?

Declaring a variable causes memory to be set aside for use by the variable. Values can then be stored in that memory space and accessed later by calling up the name given to the variable when it was declared.

Assignment of values

The equal sign in Listing 1 causes the object's reference returned by the right-hand expression to be assigned to, or saved as a value in, the reference variable named **myObjRef** *(created by the left-hand expression)* .

Memory allocation

Once the code in Listing 1 has finished execution, two distinct and different chunks of memory have been allocated and populated.

One *(potentially large)* chunk of memory has been allocated *(by the right-hand expression)* to contain the object itself. This chunk of memory has been populated according to the plans contained in the definition of the class named **Radio** .

The other chunk of memory is a relatively small chunk allocated *(by the left-hand expression)* for the reference variable containing the reference to the object.

Calling a method on the object

Assume that the definition of the **Radio** class defines a method with the following format *(also assume that this method is intended to simulate pressing a frequency-selector button on the front of the radio)* :

public void playStation(int stationNumber)

What does this mean?

Generally, in our radio-object context, this format implies that the behavior of the method named **playStation** will cause the specific station identified by an integer value passed as **stationNumber** to be selected for play.

Public and void

The *void* return type means that the method doesn't return a value.

The *public* modifier means that the button can be pressed by anyone in the car who can reach it.

(Car radios don't have frequency-selector buttons corresponding to the private modifier in Java.)

The method signature

Continuing with our exposure of jargon, some authors would say that the following constitutes the *method signature* for the method identified above:

playStation(int stationNumber)

A little more Java code

Listing 2 shows the code from the earlier listing, expanded to cause the method named **playStation** to be called.

Listing 2 . Calling the playStation method.

```
Radio myObjRef = new Radio();

myObjRef.playStation(3);
```

The first statement in Listing 2 is a repeat of the statement from the earlier listing. It is repeated here simply to maintain continuity.

Method invocation syntax

The second statement in Listing 2 is new.

This statement shows the syntax used to send a message to a Java object, or to call a method on that object *(depending on whether you prefer OOP-speak or Java-speak)* .

Join the method name to the reference

The syntax required to call a method on a Java object joins the name of the method to the object's reference, using a period as the joining operator.

*(In this case, the object's reference is stored in the reference variable named **myObjRef** . However, there are cases where an object's reference may be created and used in the same expression without storing it in a reference variable. We often refer to such an object as an anonymous object.)*

Pressing a radio button

Given the previous discussion, the numeric value 3, passed to the method when it is called, simulates the pressing of the third button on the front of the radio *(or the fourth button if you elect to number your buttons 0, 1, 2, 3, 4, 5)* .

Summary

This is the first in a miniseries of modules that describe and discuss the necessary and most significant *(essential)* aspects of OOP using Java.

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module has concentrated on encapsulation. Encapsulation was used as a springboard for a discussion of objects.

A description of an object-oriented program was provided, along with a description of an object, and how it relates to encapsulation.

In order to relate object-oriented programming to the real world, a car radio was used to illustrate and discuss several aspects of software objects.

You learned that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You learned that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value,

or some combination of the above.

You learned some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You learned where objects come from, and you learned that a class is a set of plans that can be used to construct objects. You learned that a Java object is an instance of a class.

You saw a little bit of Java code, used to create an object, and then to send a message to that object (call a method on the object).

You learned about Java references and reference variables. You learned a little about memory allocation for objects and variables in Java.

What's next?

The next module in the miniseries will introduce you to the java class.

Continuing with the real-world example introduced in this module, the next module will provide a complete Java program that simulates the manufacture and use of a car radio.

Along the way, you will see examples of *(or read about)* class definitions, constructing objects, saving references to objects, setter methods, sending messages to objects, instance variables and methods, class variables, array objects, persistence, and objects performing actions.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Objects and Encapsulation
- File: Jbs1010.htm

- Published: 08/12/14
- Revised: 09/28/15

Note: Disclaimers:
Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.
In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receives compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.
Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jbs1010-Objects and Encapsulation
Baldwin kicks off an accessible miniseries covering the necessary and most significant aspects of OOP using Java. He begins with encapsulation and objects.

Table of Contents

Preface

This module is one in a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java with particular emphasis on accessibility for blind students.

This module, along with several modules that follow provide the background information that you will need to understand the sound and audio programming material later in the course. You will need to study this material if you are new to Java programming.

On the other hand, if you already have Java programming experience, or if you are just curious, you may want to jump ahead to the module titled *Jbs2000-What is Sound?*

Prerequisites

As mentioned in an earlier module, in addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in this and the following modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ .
- A refreshable Braille display capable of providing line by line tactile output of information displayed on the computer monitor is recommended. Such a display is described at http://www.userite.com/ecampus/lesson1/tools.php , is recommended.
- The Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/8/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in this and the following modules include:

- A cursory understanding of algebra.
- An understanding of the material covered in the *Programming Fundamentals* modules that you will find in two formats at the following URLs. These modules provide fundamental programming concepts using the Java programming language in a format that should be accessible.
    - http://cnx.org/content/m45179/latest/?collection=col11441/latest
    - http://cnx.org/contents/fb64661c-5b3f-4ea8-97c6-e48df112438a

The essence of OOP

My dictionary provides several definitions for the word essence. Among those definitions are the following:

- The property necessary to the nature of a thing
- The most significant property of a thing

Thus, this and several modules that follow describe and discuss the necessary and most significant aspects of OOP using Java -- the essence of OOP using Java. For the first few modules, I will provide that information in a high-level format, devoid of any requirement to understand detailed Java syntax. In those cases where an understanding of Java syntax is required, I will provide the necessary syntax information in the form of supplementary notes.

If you understand the fundamentals of computer programming described above , you should be able to read and understand the modules in this miniseries.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

Preview

Three important concepts

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module will concentrate on encapsulation. Encapsulation will be used as a springboard for a discussion of objects.

A description of an object-oriented program will be provided, along with a description of an object, and how it relates to encapsulation.

A car radio

I suspect that many of you grew up in a family where the family car contained a push-button radio and I also suspect that you learned how to operate that radio. In fact, as children, many of you may have quarreled with your siblings about which button to push and which radio station to listen to.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects.

What you will learn

For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects. You will learn that a Java object is an *instance* of a class.

You will see a little bit of Java code, used to create an object, and then to send a message to that object *(call a method on the object)* .

You will learn about Java references and reference variables. You will also learn a little about memory allocation for objects and variables in Java.

Discussion and sample code

Purpose of the miniseries

As mentioned earlier, I will describe and discuss the necessary and most significant aspects of OOP using Java.

The three pillars

Most books on OOP will tell you that in order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Begin with encapsulation

Generally, speaking, these three concepts increase in difficulty going down the list from top to bottom. Therefore, I will begin with Encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program?

Many authors would answer this question something like the following:

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object?

An object in OOP is a software construct that encapsulates data, *(along with the ability to use or modify that data)* , into a software entity.

What is encapsulation?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

"Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface.

The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. However, I will try to illustrate it in the paragraphs that follow.

A real-world analogy

Abstract concepts, *(such as the concept of an object or encapsulation)* , can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the push-button radio in a car.

The ability to store data

Most car radios have the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data*

through use of the human interface that is provided by the manufacturer of the radio.)

The data that can be stored in a car radio includes a list of five or more frequencies that correspond to your favorite radio stations.

Using the stored data

The radio provides a mechanism *(human interface)* that allows you to use the data stored therein.

When you press one of the frequency-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)*

If you have previously stored a favorite frequency in the storage location corresponding to that button, pressing the button *(sending the message)* will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency.
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

Please change your state

By following this procedure, I "send a message" to the radio object asking it to "change its state". The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. *(Alternately, we might say that the radio object sent a message back to me.)*

We say that an object has changed its state when one or more data values stored in the object have been modified.

We also say that when an object responds to a message, it will usually perform an action, change its state, return a value, or some combination of the above.

Please perform an action

Following this, when I press that button *(send a message)* , the radio object will be automatically tuned to that frequency.

Note: Historical note:
While the ability to cause your car radio to remember your list of favorite stations may seem like a miracle of modern digital electronics, the truth is that radios had this capability long before they contained digital electronics. My first car had a radio that accomplished this feat using strings, pulleys, and levers. In fact, the radio in the first car that I owned many years ago is the first programmable device of any complexity that I can remember.
As I recall, in order to set the frequency for a button, I had to manually tune the radio to a station by turning a knob, pull one of the buttons out about a quarter of an inch, and then push it in again. From that point until I did the same thing again, whenever I pressed that button, some kind of a mechanical contraption caused a big rotary capacitor to turn just the right amount to tune for a particular radio station.
Also, I remember my grandfather having a table-model radio in the early 1940's that had radio buttons. He used them to select his favorite stations, as he surfed the airwaves.

(Interestingly, the term radio button has now become a part of programming jargon, signifying certain visual components used in graphical user interfaces.)

Enough of that, now back to my modern car radio

If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set* the frequency value associated with that button to correspond to one of the radio stations in Dallas. *(Again, I would be sending a message to the radio object asking it to change its state.)*

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the *state* of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

Note: Identical objects with identical states:
It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the frequency-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action *(tune to a particular station)* . That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Calling a method

Java-speak is a little more specific than general OOP-speak. In Java-speak, we might say that pressing one of the selector buttons on the front of the radio *calls a method* on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Java is to cause that object to execute one of its methods.

Similarly, we might say that storing a new frequency that corresponds to a particular button calls a *setter* method on the radio object.

(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)

Behavior

In addition to state, objects are often also said to have *behavior* . The overall behavior of an object is determined by the combined behaviors of its individual methods.

For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers.

Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from?

In order to mass-produce car radios, someone must first create a set of plans, *(drawings, or blueprints)* for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A class definition is a set of plans

The same is true for software objects. In order to create a software object in Java, it is necessary for someone to first create a plan.

In Java, we refer to that plan as a *class* .

The class is defined by a Java programmer. Once the class definition is available, that programmer, *(or other programmers)* , can use it to produce millions of nearly identical objects.

(While millions may sound like a lot of objects, I'm confident that since Java was released into the programming world around 1997, Java programmers around the world have created millions of objects using the standard Java class named ***Button*** *.)*

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a brand new radio and say that it is an *instance* of the

plans used to produce the radio. *(Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.)*

However, it is common jargon to refer to a software object as an *instance of a class* .

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

A little bit of code

It is time to view a little bit of Java code.

Assuming that you have access to a class definition, there are several different ways that you can create an object in Java. The most common way is using syntax similar to that shown in Listing 1 below.

Listing 1 . Instantiating a new Radio object.

```
Radio myObjRef = new Radio();
```

What does this mean?

Technically, the expression on the right-hand side of the equal sign in Listing 1 applies the ***new*** operator to a *constructor* for the class named **Radio** in order to cause the new object to come into being and to occupy memory.

(Suffice it at this point to say that a constructor is code that assists in the creation of an object according to the plans contained in a class definition. The primary purpose of a constructor is to provide initial values for the new object, but the constructor is not restricted to that behavior alone.)

A reference to the object

The right-hand expression in Listing 1 returns a *reference* to the new object.

What can you do with a reference?

The reference can later be used to send messages to the new object *(call methods belonging to the new object)* .

Saving the reference

In order to use the reference later, it is necessary to save it for later use.

The expression on the left-hand side of the equal sign in Listing 1 declares a variable of the type **Radio** named **myObjRef** .

(Because this type of variable will ultimately be used to store a reference to an object, we often refer to it by the term ***reference variable*** *.)*

What does this mean?

Declaring a variable causes memory to be set aside for use by the variable. Values can then be stored in that memory space and accessed later by calling up the name given to the variable when it was declared.

Assignment of values

The equal sign in Listing 1 causes the object's reference returned by the right-hand expression to be assigned to, or saved as a value in, the reference variable named **myObjRef** *(created by the left-hand expression)* .

Memory allocation

Once the code in Listing 1 has finished execution, two distinct and different chunks of memory have been allocated and populated.

One *(potentially large)* chunk of memory has been allocated *(by the right-hand expression)* to contain the object itself. This chunk of memory has been populated according to the plans contained in the definition of the class named **Radio** .

The other chunk of memory is a relatively small chunk allocated *(by the left-hand expression)* for the reference variable containing the reference to the object.

Calling a method on the object

Assume that the definition of the **Radio** class defines a method with the following format *(also assume that this method is intended to simulate pressing a frequency-selector button on the front of the radio)* :

public void playStation(int stationNumber)

What does this mean?

Generally, in our radio-object context, this format implies that the behavior of the method named **playStation** will cause the specific station identified by an integer value passed as **stationNumber** to be selected for play.

Public and void

The *void* return type means that the method doesn't return a value.

The *public* modifier means that the button can be pressed by anyone in the car who can reach it.

(Car radios don't have frequency-selector buttons corresponding to the private modifier in Java.)

The method signature

Continuing with our exposure of jargon, some authors would say that the following constitutes the *method signature* for the method identified above:

playStation(int stationNumber)

A little more Java code

Listing 2 shows the code from the earlier listing, expanded to cause the method named **playStation** to be called.

Listing 2 . Calling the playStation method.

```
Radio myObjRef = new Radio();

myObjRef.playStation(3);
```

The first statement in Listing 2 is a repeat of the statement from the earlier listing. It is repeated here simply to maintain continuity.

Method invocation syntax

The second statement in Listing 2 is new.

This statement shows the syntax used to send a message to a Java object, or to call a method on that object *(depending on whether you prefer OOP-speak or Java-speak)* .

Join the method name to the reference

The syntax required to call a method on a Java object joins the name of the method to the object's reference, using a period as the joining operator.

*(In this case, the object's reference is stored in the reference variable named **myObjRef** . However, there are cases where an object's reference may be created and used in the same expression without storing it in a reference variable. We often refer to such an object as an anonymous object.)*

Pressing a radio button

Given the previous discussion, the numeric value 3, passed to the method when it is called, simulates the pressing of the third button on the front of the radio *(or the fourth button if you elect to number your buttons 0, 1, 2, 3, 4, 5)* .

Summary

This is the first in a miniseries of modules that describe and discuss the necessary and most significant *(essential)* aspects of OOP using Java.

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module has concentrated on encapsulation. Encapsulation was used as a springboard for a discussion of objects.

A description of an object-oriented program was provided, along with a description of an object, and how it relates to encapsulation.

In order to relate object-oriented programming to the real world, a car radio was used to illustrate and discuss several aspects of software objects.

You learned that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You learned that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value,

or some combination of the above.

You learned some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You learned where objects come from, and you learned that a class is a set of plans that can be used to construct objects. You learned that a Java object is an instance of a class.

You saw a little bit of Java code, used to create an object, and then to send a message to that object (call a method on the object).

You learned about Java references and reference variables. You learned a little about memory allocation for objects and variables in Java.

What's next?

The next module in the miniseries will introduce you to the java class.

Continuing with the real-world example introduced in this module, the next module will provide a complete Java program that simulates the manufacture and use of a car radio.

Along the way, you will see examples of *(or read about)* class definitions, constructing objects, saving references to objects, setter methods, sending messages to objects, instance variables and methods, class variables, array objects, persistence, and objects performing actions.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Objects and Encapsulation
- File: Jbs1010.htm

- Published: 08/12/14
- Revised: 09/28/15

Note: Disclaimers:
Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.
In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receives compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.
Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jbs1020-Classes
Baldwin concentrates on a discussion of the Java class. He shows you how to write a very simple program that simulates the manufacture and use of the car radio introduced in an earlier module.

Table of Contents

Preface

This module is one in a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java with particular emphasis on accessibility for blind students.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Screen output.

Listings

Preview

This module will concentrate primarily on a discussion of the Java class.

A simple Java program will be discussed to illustrate the definition and use of two different classes. Taken in combination, these two classes simulate the manufacture and the use of the car radio object discussed in an earlier module.

You will see how to write code to create a new **Radio** object by applying the **new** operator to the class named **Radio** . You will also see how to save that object's reference in a reference variable of type **Radio** .

You will see how to write code that is used to simulate the association of a radio button with a particular radio station.

You will see how to write code that is used to simulate the pressing of a radio button to play the radio station associated with that button.

You will see the definition of a class named **Radio01** . This class consists simply of the **main** method. The **main** method of a Java application is

executed by the Java Virtual Machine when the application is run. Thus, it is the driver for the entire application.

You will see the definition of a class named **Radio** . This class includes one instance variable and two instance methods.

(The instance variable is a reference variable that refers to a special kind of object that I refer to as an array object. I will provide a very brief discussion on array objects in this module. I will have more to say about array objects in a subsequent module.)

I will provide a short discussion of class variables, which are not used in this program. I will explain that the use of class variables can often lead to undesirable side effects.

Finally, I will provide a very brief discussion of the syntax of a simple class definition in Java.

Discussion and sample code

What is a class?

I explained in an earlier module that a class is a plan from which many objects can be created. I likened the class definition to the plans from which millions of nearly identical car radios can be produced.

A simple Java program

In order to help you to get started on the right foot, and in support of future discussions, it will be advantageous to provide and discuss a simple Java program in this module.

The car radio example

Listing 9 , near the end of this module, shows the code for a simple Java application that simulates the manufacture and the use of a car radio.

Explain in fragments

In order to help you to focus specifically on important sections of code, I will explain the behavior of this program in fragments.

Top-level classes

This program contains two top-level class definitions. *(Java also supports inner classes in addition to top-level classes. Inner classes will be explained in detail in subsequent modules in this series.)*

The class named Radio01

The definition of a class named **Radio01** , is shown in its entirety in Listing 1 . The other class named **Radio** will be discussed later.

Listing 1 . The class named Radio01.

```
public class Radio01{
  public static void main(
                          String[] args){
    Radio myObjRef = new Radio();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
  }//end main
}//end class Radio01
```

The class named **Radio01** consists simply of the **main** method. The **main** method of a Java application is executed by the Java Virtual Machine when the application is run. Thus, it is the driver for the entire application.

The driver class

The code in Listing 1 simulates the manufacturer of the radio and the use of the radio by the end user. Without getting into a lot of detail regarding Java syntax, I will further subdivide and discuss this code in the following listings.

Constructing a Radio object

As discussed in an earlier module, the code in Listing 2 applies the **new** operator to the constructor for the **Radio** class, causing a new object to be created according to the plans specified in the class named **Radio** .

Listing 2 . Constructing a Radio object.

```
Radio myObjRef = new Radio();
```

Saving a reference to the Radio object

Also as discussed in an earlier module, the code in Listing 2 declares a reference variable of type **Radio** and stores the new object's reference in that variable.

Programming the radio buttons

The code in Listing 3 is new to this discussion. This statement simulates the process of associating a particular radio station with a particular button - programming a button on the radio.

As I explained in an earlier module, this is accomplished for my car radio by manually tuning the radio to a desired station and then holding the radio button down until it beeps. You have probably done something similar to this to the radio in your family's car.

Listing 3 . Programming the radio buttons.

```
myObjRef.setStationNumber(3, 93.5);
```

The statement in Listing 3 accomplishes the association of a simulated button to a simulated radio station by calling the method named **setStationNumber** on the reference to the **Radio** object. *(Recall that this sends a message to the object asking it to change its state.)*

The parameters passed to the method cause radio button number 3 to be associated with the frequency 93.5 MHz. *(The value 93.5 is stored in the variable that represents button number 3.)*

Sending a message to the object

Using typical OOP jargon, the statement in Listing 3 sends a message to the **Radio** object, asking it to change its state according to the values passed as parameters.

Pressing a button on the radio

Finally, the code in Listing 4 calls the method named **playStation** on the **Radio** object, passing the integer value 3 *(the button number)* as a parameter.

Listing 4 . Pressing a button on the radio.

```
myObjRef.playStation(3);
```

Another message

This code sends a message to the object asking it to perform an action. In this case, the action requested by the message is:

- Tune yourself to the frequency previously associated with button number 3
- Play the radio station that you find at that frequency through the speakers

How does this simulated radio play?

This simple program doesn't actually play music. As you will see later, this causes the message shown in Figure 1 to appear on the computer screen, simulating the selection and playing of a specific radio station.

Figure 1 . Screen output.

```
Playing the station at 93.5 Mhz
```

The Radio class

Listing 5 shows the class definition for the **Radio** class in its entirety.

Listing 5 . The Radio class .

Listing 5 . The Radio class .

```
class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  protected double[] stationNumber =
                           new double[5];

  public void setStationNumber(
                int index,double freq){
    stationNumber[index] = freq;
  }//end method setStationNumber

  public void playStation(int index){
    System.out.println(
             "Playing the station at "
                + stationNumber[index]
                + " Mhz");
  }//end method playStation

}//end class Radio
```

Note that the code in Listing 5 does not contain an explicit constructor. If you don't define a constructor when you define a new class, a default version of the constructor is provided on your behalf. That is the case for this simple program.

(Constructors will be explained in detail in subsequent modules.)

The plans for an object

The code in Listing 5 provides the plans from which one or more objects that simulate physical radios can be constructed.

An object instantiated *(an object is an instance of a class)* from the code in Listing 5 simulates a physical radio. I will subdivide this code into fragments and discuss it in the following listings.

An instance variable

In an earlier module, I explained that we often say that an object is an instance of a class. *(A physical radio is one instance of the plans used to produce it.)* The code in Listing 6 shows the declaration and initialization of what is commonly referred to as an instance variable.

Listing 6 . An instance variable.

```
protected double[] stationNumber =
                          new double[5];
```

Why call it an instance variable?

The name *instance variable* comes from the fact that every instance of the class *(object)* has one. *(Every radio produced from the same set of plans has the ability to associate a frequency with each selector button on the front of the radio.)*

Class variables - an aside

Note that Java also supports something called a *class variable* , which is different from an *instance variable* .

Class variables are shared among all of the objects created from a given class. Stated differently, no matter how many objects are instantiated from a class definition, they all share a single copy of each class variable.

There is no analogy to a class variable in a physical radio object. Radios are installed in different cars separated from each other by thousands of miles. Therefore, there can be no sharing of anything among different physical radio objects.

(Well, that may not be entirely true. In today's technology, different radio objects could potentially share something at a common location via satellite communications, but my car radio doesn't do anything like that.)

Class variables can cause undesirable side effects

While class variables are relatively easy to use in Java, they are difficult to explain from an OOP viewpoint. Also, it is my opinion that from a good overall design viewpoint, class variables should be used very sparingly, if at all.

Therefore, for the first several modules, I will exclude the possibility of class variables in this series of modules. *(I will explain the use of class variables in Java in a subsequent module.)*

Reference to an array object

Now, let's get back to the instance variable named **stationNumber** shown in Listing 6 . Without getting into a lot of detail, this variable is also a reference variable, referring to an array object.

The array object encapsulates a simple one-dimensional array with five elements of type **double** . *(Java array indices begin with zero, so the index values for this array extend from 0 to 4 inclusive. I will also discuss array objects in more detail in a subsequent module.)*

Persistence

The array object is where the data is stored that associates the frequency of a radio station with the simulated physical button on the front of the radio.

Each element in the array corresponds to one frequency-selector button on the front of the radio. Hence, the radio simulated by an object of the **Radio**

class has five simulated frequency-selector buttons.

The array object exists when the code in Listing 6 has finished executing. Each element in the array has been initialized to a default value of 0.0 *(double-precision float value of zero)* .

The setStationNumber method

Listing 7 shows the **setStationNumber** method in its entirety

Listing 7 . The setStationNumber method.

```
public void setStationNumber(
                int index,double freq){
  stationNumber[index] = freq;
}//end method setStationNumber
```

Associates radio station with button

This is the method that is used to simulate the behavior of having the user associate a particular button with a particular radio station. *(Recall that this is accomplished on my car radio by manually tuning the radio to a specific station and then holding the button down until it beeps. Your family's car radio probably operates in some similar way.)*

This method receives two incoming parameters:

- An integer that corresponds to a button number *(button numbers are assumed to begin with 0 and extend through 4 in order to match array indexes)*
- A frequency value to be associated with the indicated button.

Save the frequency value

The code in the method stores the frequency value in an element of the array object discussed earlier.

The element number is specified by the value of index shown in square brackets in the assignment expression. *(This syntax is similar to storing a value in an array element in most programming languages that I am familiar with.)*

Pressing a radio button to select a station

Listing 8 shows the **playStation** method. This is the method that simulates the result of having the user press a button on the front of the radio to select a particular radio station for play.

Listing 8 . The playStation method.

```
public void playStation(int index){
  System.out.println(
          "Playing the station at "
             + stationNumber[index]
             + " Mhz");
}//end method playStation
```

Selecting and playing a radio station

The method receives an integer index value as an incoming parameter. This index corresponds to the number of the button pressed by the user. This method simulates the playing of the radio station by

- Extracting the appropriate frequency value from the array object, and
- Displaying that value on the computer screen along with some surrounding text.

When called by code in the **main** method of this program, this method produces the message shown in Figure 1 on the computer screen

That summarizes the behavior of this simple program.

Class definition syntax

There are a number of items that can appear in a class definition, including the following:

- Instance variables
- Class variables
- Instance methods
- Class methods
- Constructors
- Static initializer blocks
- Inner classes

Let's keep it simple

In order to make these modules as easy to understand as possible, the first several modules will ignore the possibility of class variables, class methods, static initializer blocks, and inner classes.

As mentioned in the earlier discussion of class variables, these elements aren't particularly difficult to use, but they create a lot of complications when attempting to explain OOP from the viewpoint of Java programming.

Therefore, the first several modules in the series will assume that class definitions are limited to the following elements:

- Instance variables
- Instance methods
- Constructors

A constructor

A constructor is used only once in the lifetime of an object. It participates in the task of creating *(instantiating)* and initializing the object. Following instantiation, the state and behavior of an object depends entirely on instance variables, class variables, instance methods, and class methods.

Instance variables and methods

The class named **Radio** discussed earlier contains

- One instance variable named **stationNumber** , and
- Two instance methods named **setStationNumber** and **playStation** .

Summary

This module has concentrated primarily on a discussion of the Java class.

A simple Java program was discussed to illustrate the definition and use of two different classes. Taken in combination, these two classes simulate the manufacture and use of the car radio object introduced in an earlier module.

You saw how to write code to create a new **Radio** object by applying the **new** operator to the class named **Radio** .

You also saw how to save that object's reference in a reference variable of type **Radio** .

You saw how to write code *(in an instance method named **setStationNumber**)* used to simulate the association of a radio button with a particular radio station.

You saw how to write code *(in an instance method named **playStation**)* to simulate the pressing of a radio button to play the radio station associated with that button.

You saw the definition of the class named **Radio01** , which consists simply of the **main** method. The **main** method of a Java application is executed by

the Java Virtual Machine when the application is run.

You saw the definition of the class named **Radio** . This class includes one instance variable and two instance methods. *(The instance variable is a reference variable that refers to a special kind of object that I refer to as an array object. I provided a very brief discussion on array objects. I will have more to say on this topic in a subsequent module.)*

I provided a short discussion of class variables, which are not used in this program. I explained that the use of class variables can often lead to undesirable side effects.

Finally, I provided a very brief discussion of the syntax of a simple class definition in Java.

What's next?

Recall that in order to understand OOP, you must understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

The next module will begin a discussion of inheritance. Overall, the discussion of inheritance will require more than one module. In the next module, I will discuss how the definition of a class defines a new data type. I will show you how to extend an existing class. I will explain what is inherited through inheritance. I will discuss code reuse and explicit constructors.

Finally, I will illustrate all of the above in a program that produces an audio output.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs1020-Classes
- File: Jbs1020.htm
- Published: 08/12/14
- Revised: 09/28/15

Complete program listing

Listing 9 provides a complete listing of the program named **Radio01** .

Listing 9 . The program named Radio01.

```
/*File Radio01.java
Copyright 2001, R.G.Baldwin
Simulates manufacture and use of a
car radio.

This program produces the following
output on the computer screen:

Playing the station at 93.5 Mhz
**************************************/

public class Radio01{
  //This class simulates the
  // manufacturer and the human user
  public static void main(
                          String[] args){
    Radio myObjRef = new Radio();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
  }//end main
}//end class Radio01
  //--------------------------------//

class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  protected double[] stationNumber =
                           new double[5];

  public void setStationNumber(
                 int index,double freq){
    stationNumber[index] = freq;
  }//end method setStationNumber
```

Listing 9 . The program named Radio01.

```
  public void playStation(int index){
    System.out.println(
            "Playing the station at "
               + stationNumber[index]
               + " Mhz");
  }//end method playStation

}//end class Radio
```

-end-

Jbs1050-A Roadmap
This module provides a roadmap for blind students to navigate through a large number of Object-Oriented Programming (OOP) modules for the purpose of learning OOP using Java.

Table of contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . The module provides a roadmap for blind students to navigate through a large number of (mostly) accessible Object-Oriented Programming (OOP) modules for the purpose of learning OOP using Java. *(If you discover accessibility problems in those modules, please let me know and I will try to fix them.)*

Discussion

A small taste of OOP

By studying the following three modules, you have already gotten a small taste of what OOP is all about:

- Jbs1000-Getting Started
- Jbs1010-Objects and Encapsulation
- Jbs1020-Classes

Now it's time to jump in with both feet and really learn what OOP is all about. The purpose of this module is to provide a roadmap to help you navigate through a large number of (mostly) accessible Object-Oriented Programming (OOP) modules for the purpose of learning OOP using Java.

Programming fundamentals

You should begin by studying the following modules on **Programming Fundamentals** . These modules assume that you are starting from ground zero with no programming knowledge. If you already have some programming knowledge, you may be able skip some of these modules, or at least work through them very quickly. One good way to confirm the level of your programming knowledge is to work through the self-assessment tests that I will discuss later.

- Jb0103 : Preface to Programming Fundamentals
- Jb0105 : Java OOP: Similarities and Differences between Java and C++
- Jb0110 : Java OOP: Programming Fundamentals, Getting Started
- Jb0110r : Review
- Jb0115 : Java OOP: First Program
- Jb0120 : Java OOP: A Gentle Introduction to Java Programming
- Jb0120r : Review
- Jb0130 : Java OOP: A Gentle Introduction to Methods in Java
- Jb0130r : Review
- Jb0140 : Java OOP: Java comments
- Jb0140r : Review
- Jb0150 : Java OOP: A Gentle Introduction to Java Data Types
- Jb0150r : Review
- Jb0160 : Java OOP: Hello World
- Jb0160r : Review
- Jb0170 : Java OOP: A little more information about classes

- Jb0170r : Review
- Jb0180 : Java OOP: The main method
- Jb0180r : Review
- Jb0190 : Java OOP: Using the System and PrintStream Classes
- Jb0190r : Review
- Jb0200 : Java OOP: Variables
- Jb0200r : Review
- Jb0210 : Java OOP: Operators
- Jb0210r : Review
- Jb0220 : Java OOP: Statements and Expressions
- Jb0220r : Review
- Jb0230 : Java OOP: Flow of Control
- Jb0230r : Review
- Jb0240 : Java OOP: Arrays and Strings
- Jb0240r : Review
- Jb0250 : Java OOP: Brief Introduction to Exceptions
- Jb0260 : Java OOP: Command-Line Arguments
- Jb0260r : Review
- Jb0270 : Java OOP: Packages
- Jb0280 : Java OOP: String and StringBuffer
- Jb0280r : Review
- Jb0290 : The end of Programming Fundamentals

Self-assessment tests

Earlier I mentioned the availability of a set of self-assessment tests. If you already have some programming knowledge, you should take the following tests before beginning the Programming fundamentals material to assess the level of your knowledge. That will help you decide which, if any, of the Programming fundamentals modules you need to study.

If you don't already have programming knowledge, you should begin with Programming fundamentals and use the self-assessment tests to determine how well you are learning that material. In that case, I recommend that you begin taking the self-assessment tests after you complete Jb0180 and

continue studying the "fundamentals" material and taking the self-assessment tests in parallel.

- Ap0005 : Preface to OOP Self-Assessment
- Ap0010 : Self-assessment, Primitive Types
- Ap0020 : Self-assessment, Assignment and Arithmetic Operators
- Ap0030 : Self-assessment, Relational Operators, Increment Operator, and Control Structures
- Ap0040 : Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method
- Ap0050 : Self-assessment, Escape Character Sequences and Arrays
- Ap0060 : Self-assessment, More on Arrays

More advanced material

By the time you reach this point, you should have the fundamentals well in hand. It is time to start studying more advanced material. You should start studying the material listed under Essence of OOP and continue using the following self-assessment tests to assess your progress along the way.

- Ap0070 : Self-assessment, Method Overloading
- Ap0080 : Self-assessment, Classes, Constructors, and Accessor Methods
- Ap0090 : Self-assessment, the super keyword, final keyword, and static methods
- Ap0100 : Self-assessment, The this keyword, static final variables, and initialization of instance variables
- Ap0110 : Self-assessment, Extending classes, overriding methods, and polymorphic behavior
- Ap0120 : Self-assessment, Interfaces and polymorphic behavior
- Ap0130 : Self-assessment, Comparing objects, packages, import directives, and some common exceptions
- Ap0140 : Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references

Essence of OOP

By studying the material under A small taste earlier, you got a small taste of what OOP is all about. By studying the material under Programming fundamentals and Self-assessment tests , you learned a lot about programming fundamentals using the Java programming language.

Now that you have a better understanding of the fundamentals, we will cover some very advanced OOP material in the following modules:

- Java1600 : Objects and Encapsulation
- Java1602 : Classes
- Java1604 : Inheritance, Part 1
- Java1606 : Inheritance, Part 2
- Java1608 : Polymorphism Based on Overloaded Methods
- Java1610 : Polymorphism, Type Conversion, Casting, etc.
- Java1612 : Runtime Polymorphism through Inheritance
- Java1614 : Polymorphism and the Object Class
- Java1616 : Polymorphism and Interfaces, Part 1
- Java1618 : Polymorphism and Interfaces, Part 2
- Java1620 : Static Members
- Java1622 : Array Objects, Part 1
- Java1624 : Array Objects, Part 2
- Java1626 : Array Objects, Part 3
- Java1628 : The this and super Keywords
- Java1630 : Exception Handling
- Java1636 : Member Classes - optional for this course
- Java1638 : Local Classes - optional for this course
- Java1640 : Anonymous Classes - optional for this course

The Java Collections Framework

Before we move on to the modules on Java sound, you will need to understand the material in the following modules:

- Java4010 : Getting Started with Java Collections

- Java4010r : Review
- Java4020 : What is a Collection
- Java4020r : Review
- Java4030 : Purpose of Framework Interfaces
- Java4030r : Review
- Java4040 : Purpose of Framework Implementations and Algorithms
- Java4040r : Review
- Java4050 : Core Collection Interfaces
- Java4050r : Review
- Java4060 : Duplicate Elements, Ordered Collections, Sorted Collections, and Interface Specialization
- Java4060r : Review
- Java4070 : The Comparable Interface, Part 1
- Java4070r : Review
- Java4080 : The Comparable Interface, Part 2
- Java4080r : Review
- Java4090 : The Comparator Interface, Part 1
- Java4090r : Review
- Java4100 : The Comparator Interface, Part 2
- Java4100r : Review
- Java4110 : The Comparator Interface, Part 3
- Java4110r : Review
- Java4120 : The Comparator Interface, Part 4
- Java4120r : Review
- Java4130 : The Comparator Interface, Part 5
- Java4130r : Review
- Java4140 : The Comparator Interface, Part 6
- Java4140r : Review
- Java4150 : The toArray Method, Part 1
- Java4150r : Review
- Java4160 : The toArray Method, Part 2
- Java4160r : Review

Java sound

By this point in the course, you should have learned enough that you can put Java OOP to work in a substantive way. Normally at this point, my sighted students would begin using OOP to manipulate the pixels and colors in digital images. Obviously, that is not something that blind students could do. In this course, which is designed specifically for blind students, you will begin writing substantive programs that deal with something that you are probably very good at -- sound. The lessons in this part of the course begin at Jbs2000-What is Sound? and continue from there.

At the beginning, you will learn to write a Java OOP program to produce simple sounds such as Tones *(click to download and play)* . If you are viewing the HTML version of this module on the Legacy site, you should be able to download this file and play it using any standard media player that supports audio files of type AU. *(If you are on the OpenStax site, or if you are viewing the PDF version of this module on either site, you may not be able to download the file.)*

As you move through the sound modules, you will learn to write OOP programs to produce more complex sounds such as StereoPingpong *(click to download and play)* .

Ultimately you will learn to write OOP programs that can be used to compose and play various melodies such as the following:

- FourScales
- MaryHadALittleLamb
- Greensleeves

Stay tuned. There is much more to come.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs1050: A Roadmap
- File: Jbs1050.cnxml.htm
- Published: 08/17/14
- Revised: 09/29/15

Note: Disclaimers:
Financial : Although the **OpenStax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
You also need to know that Prof. Baldwin receives no financial compensation from **OpenStax CNX** even if you purchase the PDF version of the collection.
In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receives compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **OpenStax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.
Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jbs2000-What is Sound?
This module explains some of the physical aspects of sound in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains some of the physical aspects of sound in a format that is accessible to blind students. It also explains a little about the trigonometric sine and cosine

functions and suggests that they can be used to create sound with a computer. An audio file is provided where audio pulses of different frequencies are used to represent points on a graph of a sinusoid.

General background information

By this point in the course, you should have learned enough that you can put Java OOP to work in a substantive way. Normally at this point, my sighted students would begin using OOP to manipulate the pixels and colors in digital images. That is not well suited to teaching OOP to blind students. In this course, which is designed specifically for blind students, you will begin writing substantive programs that deal with something that you are probably very good at -- sound.

For example, you will soon learn how to write a Java OOP program to produce simple sounds such as TonesStereo .

(Note that if you are viewing the HTML version of this module on the Legacy site, you should be able to download this file and the other audio files in this module and play them using any standard media player that supports audio files of type AU. However, if you are on the OpenStax site, or if you are viewing the PDF version of this module on either site, you may not be able to download the file.)

As you study future modules in this collection, you will learn to write OOP programs to produce more complex sounds such as StereoPingpong .

Ultimately you will learn to write OOP programs that can be used to compose and play various melodies such as the following:

- FourScales
- MaryHadALittleLamb
- Greensleeves

Discussion

What is sound ?

An old physics question goes something like this. If the earth were populated solely by humans and a tree fell in the forest with no humans present, would it make a sound?

The answer is "No, it would not make a sound." The explanation follows.

Vibration

Assume that someone strikes the top of a car with their hand. This would cause the metal roof of the car to start vibrating. That vibration would cause the adjacent air molecules to also vibrate causing them to collide with their neighbors. Those molecules would collide with their neighbors, and on and on. We can think of this as a wave of vibrating molecules propagating outward from the car.

This wave of vibrating molecules propagates outward at a speed of approximately one mile every five seconds and decreases in intensity as it propagates out from the source. At some distance from the source, the intensity of the vibrations has decreased to an insignificant level.

Sensing vibration

If you are standing somewhere between the car and the point at which the vibrations are no longer significant, the air molecules in your ear canals will begin to vibrate. This will stimulate nerves that lead from your ear to your brain. Your brain will interpret the stimulation as something that we refer to *sound* .

Therefore, when the tree falls, it causes the air molecules to vibrate. However, if there are no ear canals around, no nerves to be stimulated, and no brain to interpret that stimulation, there is no sound.

Creating sound with a hammer

All we need to do to cause a person to experience *sound* is to cause the air molecules in that person's ear canals to vibrate at a frequency somewhere between 20 Hz *(cycles per second)* and 20,000 Hz, depending on the person's age. Usually as a person ages, the range defined by those upper and lower limits decreases. Thus, a young person may "hear" the high-pitched whine of a power saw while an older person may not experience sound in that situation.

There are many ways that we can cause most humans to experience sound. One way for example, is to strike a bell with a hammer. Another way is to write a program to cause the diaphragm of a computer *speaker* to vibrate. That will be the topic of the next few modules in this course.

Creating sound with a computer

One way to create sound with a computer would be to hold it out and drop it on the floor. However, that is not what we intend to do. Instead, we intend to write computer program that will cause the computer to create sound.

An object of a class

As you will learn in a future module, we can define a class and instantiate an object of that class for use in producing sound with a computer. When used properly in a program, that object will cause signed numeric values stored in an array to be converted to electrical current in the speakers attached to the computer. Positive values will cause the current to flow in one direction and negative values will cause the current to flow in the opposite direction.

Alternating positive and negative values

If we populate that array with alternating groups of positive and negative values, that will cause the current being delivered to the speakers to

alternate, switching from one direction to the other. The level of the alternating currents will be roughly proportional to the magnitudes of the alternating positive and negative values. Given sufficient magnitude within a frequency range supported by the speakers, the alternating currents will cause the diaphragms of the speakers to vibrate or move back and forth. This, in turn, will cause the adjacent air molecules to vibrate, which may be sensed as sound by a person within hearing distance of the speakers.

Sine and cosine functions

Populating the array

There are many ways that we can write a program to populate the array with alternating groups of positive and negative numbers. One of the simplest ways is to populate the array with scaled versions of the values produced by the static **sin** function or the static **cos** function of the **Math** class. The **Math** class is contained in the Java Standard Edition class library. A common name for a series of such values is *sinusoid* .

In addition to being one of the easiest ways to populate the array, this is also a way to produce a sound that is very close to being a *pure tone* . By pure tone, I mean a sound that results from something vibrating at a single frequency such as Middle-C *((261.63 Hz)* .

The shape of a sinusoid

If you have taken a math course in trigonometry, you probably know the shape produced by graphing the output from either the **sin** function or the **cos** function. If not, it would be good if you can ask a sighted friend to Google for "sinusoidal function image", download and print one or more of the images and emboss them for you using whatever embossing method you prefer. This will provide you with a tactile representation of a sinusoid.

An audio graph of a sinusoid

In any event, I am going to provide you with a sound file named SinusoidalAudioGraph that provides an audio representation of the graph of a sinusoid.

This audio file contains an 8-second melody consisting of 32 uniformly spaced pulses at different frequencies. The frequencies *(pitches)* of the pulses are centered on middle-C *(261.63 Hz)* . The frequency deviation from middle-C versus time is based on a sinusoidal function with a frequency of 0.5 Hz.

Points on graph

Each pulse represents one point on a graph of the sinusoid. Pulses with frequencies at or above middle-C are delivered to the left speaker. Pulses with frequencies below middle-C are delivered to the right speaker.

The audio output can be thought of as an audio representation of a graph of a sinusoid. Pulses with frequencies above middle-C represent points on the positive lobe of the sinusoid. Increasing pitch represents increasing amplitude on the graph of the sinusoid.

Pulses with frequencies below middle-C can be thought of as representing points on the negative lobe of the sinusoid. In this case, decreasing pitch represents points on the sinusoid that are further from the horizontal axis in the negative direction.

Pulses with a frequency of middle-C can be thought of as representing points on the horizontal axis with a value of zero.

Four complete cycles of the 0.5 Hz sinusoid are represented by the 32 pulses in the 8-second melody.

Hopefully, by listening to this audio file, you can get an idea of the shape of a sinusoid.

A dual purpose

In addition to providing the audio file to help you discern the shape of a sinusoid, I am also providing it as an example of the type of audio files that you will be able to create once you understand the programs in upcoming modules.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2000-What is Sound?
- File: Jbs2000.htm
- Published: 08/25/14
- Revised: 09/29/15

copy of a collection that is freely available on **OpenStax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.
Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jbs2010-Your First Sound Program
This module explains how to write a program that creates an audio output consisting of white noise in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains how to write a program that creates an audio output consisting of white noise.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

General background information

By this point in the course, you should have learned enough that you can put Java OOP to work in a substantive way. In this course, which is designed specifically for blind students, you will write substantive programs that deal with something that you are probably very good at -- sound.

An earlier module explained some of the physical aspects of sound. It also explained a little about how you can write programs to create sound with a computer.

This module will show you how to write a Java OOP program that uses a random number generator to produce *white noise* .

(White noise is usually thought to be noise that contains equal contributions of all frequencies. Pink noise is thought of as noise that contains almost equal contributions of almost all frequencies. In reality, the program that we will examine in this module will probably produce pink noise instead of white noise.)

Download and play the file named WhiteNoise to hear what white or pink noise sounds like. You will probably conclude that it isn't very pleasant to listen to. You should be able to play the audio file with any standard media player that can handle the AU file type. *(In case you are on the OpenStax site and you are unable to download the audio file, click the **Legacy Site** link at the top of this page to switch over to the same module on the Legacy site. You should be able to download the audio file from there.)*

Discussion and sample code

The class named AudioPlayOrFile01

A formal mechanism for code reuse

One of the most important aspects of OOP is that it provides a formal mechanism for code reuse without a requirement for the distribution of source code. For example, if some other programmer

- has developed a class *(or a group of classes)* that does something that you need to do, and
- if you have confidence in that programmer, and
- if that programmer is willing to provide you with the compiled files for those classes, then
- you can use them in your program without knowing or caring how they are implemented.

All you need is documentation on how to use the classes.

The Java Standard edition library

The Java Standard edition library contains thousands of compiled class files that are available for every Java programmer to use whenever and wherever they are needed. Java programmers routinely use those classes every day without knowing or caring how they are actually implemented.

For example, the library contains a class named **Random** that can be used to produce a series of pseudo-random numbers. We will use that class in the program in this module. I don't know, nor do I care, how the code in that class produces the numbers. I'm confident that it behaves as advertised. Therefore, all I care about is how to use it. I can learn that from the library documentation.

Two special classes

I will provide the source code for two special classes in this module that I will use in several future modules without modification. I suggest that you simply have confidence that they behave as advertised and use them. I will provide very little, if any, explanation as to how they do what they do. Those classes are named

- **AudioPlayOrFile01**
- **AudioFormatParameters01**

The class named **AudioPlayOrFile01** accepts an array of **byte** data as input, along with some other information, and causes electrical currents to flow through audio speakers that are attached to the computer. A complete listing of the class named **AudioPlayOrFile01** is provided in Listing 12 .

This class is very simple to use. As you can see in Listing 12 , the constructor for the class requires three parameters:

- A reference to an object of the class **AudioFormatParameters01** *(see this section)* .
- A reference to an array containing elements of type **byte** , which define the sound that is to be played or filed. The format of the array will be defined later.
- A reference to an object of type **String** containing either "play", which will cause the sound to be played immediately, or containing a string that can be used as the name of an output audio file of type AU.

Code reuse

It is unlikely that the class named **AudioPlayOrFile01** will be modified throughout the next several modules. Therefore, unless I change the name of the class, *(which I will do if I modify the file)* , you will only need to copy and compile the source code once. After that, you can simply copy the compiled class file from one project folder to the next.

The class named AudioFormatParameters01

An object of the class named **AudioFormatParameters01** is simply a container for some of the parameters required by the class named **AudioPlayOrFile01** . A complete listing of the class named **AudioFormatParameters01** is provided in Listing 13 .

With the exception of **bigEndian** , the comments should be sufficient to describe the parameters. If you would like to learn more about **bigEndian** , simply Google it.

Disk organization

Although there are more elegant ways to organize your disk by making use of the *classpath* environment variable, unless stated otherwise, the disk organization that I will use in this collection of sound modules will not use the *classpath* environment variable.

With the exception of the classes in the Java Standard Edition library, these modules will assume that source code files or compiled class files for all required classes are physically contained in the folder from which the program is executed. For the program that I will explain in this module, this means that the following source code or compiled class files must be contained in the execution folder:

- MusicComposer04
- AudioSignalGenerator02
- WhiteNoise
- AudioFormatParameters01
- AudioPlayOrFile01

The class named MusicComposer04

An object of this class creates and plays three seconds of monaural *white* or *pink* noise. It works in conjunction with the following classes:

- WhiteNoise
- AudioSignalGenerator02
- AudioPlayOrFile01
- AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of type AU for playback later. *(Click WhiteNoise to download and play the audio file produced by this program.)*

A complete listing of the class named **MusicComposer04** is provided in Listing 14 . I will break this code down and explain it in fragments. The first fragment is shown in Listing 1 .

Beginning of class MusicComposer04

Listing 1 shows the beginning of the class named **MusicComposer04** along with the **main** method.

Listing 1 . Beginning of the class named MusicComposer04.

Listing 1 . Beginning of the class named MusicComposer04.

```
public class MusicComposer04{

  AudioFormatParameters01 audioParams = new AudioFormatParameters01();
  byte[] melody;
  String[] args;
  //--------------------------------------------------------------------
-----//

  public static void main(String[] args){
    new MusicComposer04(args);
  }//end main
  //--------------------------------------------------------------------
-----//
```

The instance variables

The first instance variable named **audioParams** instantiates and saves a reference to an object containing audio format parameters with predefined values. They may be modified by the signal generator later at runtime. Values that are allowed by Java SDK 1.4.1 are shown in comments in the class definition for the class named AudioFormatParameters01 .

The instance variable named melody will ultimately refer to an array object that holds the audio data that will be played or filed.

The instance variable named args will ultimately refer to an array object that holds an incoming command-line parameter.

Command-line parameters

Only one command-line parameter is needed and it is optional. If the value of the parameter is "play", the sound will be played immediately. Otherwise, the parameter string will be used as a filename for an audio file of type AU. In this case, it must be a string that would be valid as a file name for your operating system.

The main method

The **main** method simply instantiates a new object of this class passing the incoming **args** array's reference to the constructor.

Compiling and executing the program

I typically use Windows batch files to compile and execute my Java program. Listing 2 shows the contents of such a batch file. This batch file will attempt to:

- Delete all class files in the folder.
- Delete a file named WhiteNoise.au if it exists.
- Compile the program.
- Execute the program once to play the sound immediately.
- Execute the program again to write the sound to the audio file named WhiteNoise.au.
- Delete all the class files in the folder.

Listing 2 . The file named WhiteNoise.bat.

```
echo off
del *.class

del WhiteNoise.au

echo on
javac MusicComposer04.java
java MusicComposer04 play
java MusicComposer04 WhiteNoise

echo off
del *.class

pause
```

The constructor for MusicComposer04

Listing 3 shows the beginning of the constructor for **MusicComposer04** .

Listing 3 . Beginning of the constructor for MusicComposer04.

Listing 3 . Beginning of the constructor for MusicComposer04.

```
  public MusicComposer04(String[] args){//constructor

    this.args = args;

    //Create default args data if no args data is provided on the
command line.
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if
```

The constructor begins by saving the incoming reference to the **args** array containing the command-line parameter.

If the length of the **args** array is zero, this means that the user did not enter a command-line parameter. In that case, the constructor instantiates an appropriate array object, populates its only element with a reference to a "play" string, and saves that array object's reference in the instance variable discussed earlier .

Get and play or file the sound

Listing 4 instantiates a new object of the **WhiteNoise** class and saves the object's reference in a local reference variable of type **WhiteNoise** named **whiteNoise** . You are already familiar with the three parameters that are passed to the constructor for the **WhiteNoise** class.

*(Later when we discuss runtime polymorphism, we will save such an object's reference in a reference variable of type **AudioSignalGenerator02** , which is the immediate superclass of the **WhiteNoise** class.)*

Listing 4 . Get and play or file the sound.

```
    //Get a populated array containing audio data for white or pink
noise.
    WhiteNoise whiteNoise = new
WhiteNoise(audioParams,this.args,melody);
    melody = whiteNoise.getMelody();

    //Play or file the audio data
    new
AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-----------------------------------------------------------------
-----//
}//end class MusicComposer04.java
```

Then Listing 4 calls the **getMelody** method and saves the reference that is returned by the method in the instance variable named **melody** discussed earlier .

Finally, Listing 4 instantiates an object of the AudioPlayOrFile01 class, passing **audioParams** , **melody** , and **args[0]** as parameters, and then calls the method named **playOrFileData** on that object's reference. Depending on the value of **args[0]** , this method will either play the sound contained in **melody** immediately, or write it into an output audio file.

The class named AudioSignalGenerator02

The class named **AudioSignalGenerator02** is shown in Listing 15 . This is another class that will probably be used throughout this series of modules on Java sound and which probably won't change, at least for the early modules in the series. Therefore, we will only need to discuss it this one time.

Beginning of the class named AudioSignalGenerator02

This is an abstract class that serves as the base class or superclass for several other classes that can be used to create sounds or melodies of different types.

Listing 5 . Beginning of the class named AudioSignalGenerator02.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This class can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //----------------------------------------------------------------------
-----//
```

The code in Listing 5 declares four instance variables. You are already familiar with the purpose of three of them. The variable named **byteBuffer** isn't actually used in the program in this module, but will be used in later modules. I will explain it in the first module in which it is actually used.

The constructor for AudioSignalGenerator02

The constructor is shown in Listing 6 . The constructor simply receives three input parameters and saves them in the instance variables shown in Listing 5 .

Listing 6 . The constructor for AudioSignalGenerator02.

```
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------
-----//
```

The abstract method named getMelody

Listing 7 shows an abstract method named **getMelody** . In case you don't remember what an abstract method is, see *Ap0100 : Self-assessment, The this keyword, static final variables, and initialization of instance variables* .

Briefly, an abstract method is a method that is designed to be overridden. The method named **getMelody** in Listing 7 must be overridden in a subclass to be of any use. You will see the overridden version of the method named **getMelody** shortly in the class named **WhiteNoise** .

Recall that the method was called on a reference to a **WhiteNoise** object in Listing 4 . *(You will get a better idea of the purpose of this abstract method when we discuss runtime polymorphism in a future module.)*

Listing 7 . The abstract method named getMelody.

```
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
```

The class named WhiteNoise

Beginning of the class named WhiteNoise

The sound that you heard when you listened to the audio file named WhiteNoise was actually produced by the **getMelody** method of the **WhiteNoise** class. That class begins in Listing 8 and the **getMelody** method begins in Listing 9 .

The first thing to notice about Listing 8 is that the **WhiteNoise** class extends *(is a subclass of)* the abstract **AudioSignalGenerator02** class.

The constructor for the WhiteNoise class

The constructor for the class named **WhiteNoise** , which is shown in Listing 8 , receives three incoming parameters and passes them up to the superclass named **AudioSignalGenerator02** through the use of the **super** keyword where they are saved *(see Listing 6)* . In case you don't remember the **super** keyword, see *Java1628 : The this and super Keywords*. Also see *Ap0090 : Self-assessment, the super keyword, final keyword, and static methods* .

Listing 8 . Beginning of the class named WhiteNoise.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class WhiteNoise extends AudioSignalGenerator02{

  public WhiteNoise(AudioFormatParameters01 audioParams,
                    String[] args,
                    byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
```

Beginning of the getMelody method

Listing 9 shows the beginning of the overridden **getMelody** method. *(Recall that an abstract version of this method is inherited from the class named **AudioSignalGenerator02** , see Listing 7)*

Listing 9 . Beginning of the getMelody method.

```
  byte[] getMelody(){
    //Set the audio parameters to mono
    audioParams.channels = 1;//superfluous -- default value
    System.out.println("audioParams.channels = " +
audioParams.channels);
```

Note that the **getMelody** method returns a reference to an array object containing elements of type **Byte** .

Overall structure of the program

The code in Listing 1 declares an instance variable named **melody** as a reference to an array object containing elements of type **Byte** . The code in Listing 4 calls the **getMelody** method on a reference to an object of the **WhiteNoise** class and assigns the returned reference to the instance variable named **melody** that is declared in Listing 1 .

Listing 4 passes that reference to the **playOrFileData** method to cause the contents of the array to be played through the computer speakers or saved in an audio file.

Now that you understand the overall structure, we need to examine the details of the code to determine how the method named **getMelody** actually returns a reference to an array object containing data that represents three seconds of monaural white noise.

The predefined audio parameters

Because the class named **WhiteNoise** extends the class named **AudioSignalGenerator02** , it inherits a reference to an object of the class **AudioFormatParameters01** . The name of the inherited reference is **audioParams** . As you learned earlier , this object contains several predefined audio parameters. As you can see in Listing 13 , the predefined values are all stored in public instance variables. Therefore, code in methods of the **WhiteNoise** class can modify the values stored in that object by joining the object's reference to the name of the variable as shown in Listing 9 *(* ***audioParams.channels*** for example).

The predefined value for the variable named **channels** in Listing 13 is 1 for monaural. That is the value that is needed for this monaural program. Listing 9 contains a statement that sets the value of **audioParams.channels** to 1 simply to illustrate the syntax involved. That statement is superfluous. However, as you will see in Listing 10 , there is a statement that sets the value of **audioParams.sampleRate** replacing the default value of 16000.0F with 8000.0F. That statement is not superfluous. *(Recall that the F suffix causes the value to be treated as type* ***float*** *instead of type* ***double*** *.)*

Miscellaneous setup operations

Listing 10 contains several miscellaneous setup operations. The comments should be sufficient to describe the purpose of each of those operations.

Listing 10 . Miscellaneous setup operations.

Listing 10 . Miscellaneous setup operations.

```
    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are
8000,11025,
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes
per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*

bytesPerSampPerChan*audioParams.channels)];
    System.out.println("melody.length = " + melody.length);
```

Generating the white (random) noise

Audio data consisting of a sequence of random values is believed to be white, or at least pink. The code in Listing 11 uses a class named **Random** from the Java Standard Edition library to populate the array with bytes having random values.

Listing 11 . Generating the white (random) noise.

```
    Random generator = new Random(new Date().getTime());
    for(int cnt = 0;cnt < melody.length; cnt++){
      melody[cnt] = (byte)generator.nextInt();
    }//end for loop
    return melody;
  }//end method getMelody
  //-----------------------------------------------------------------
-----//

}//end class WhiteNoise
```

A stream of pseudorandom numbers

According to the documentation, an instance of the **Random** class can be used to generate a stream of pseudorandom numbers. *(The prefix pseudo means that the numbers in the stream are almost random but may not be completely random.)*

We need a stream of pseudorandom **byte** values. The **Random** class provides methods named **nextBoolean** , **nextDouble** , **nextFloat** , **nextGaussian** , **nextInt** , and **nextLong** that can be called to get one new value from the pseudorandom stream as the type indicated by the name of the method. However, there is no method that returns the next value as type **short** or as type **byte** . *(There is a method named **nextBytes** -- note the plural. It behaves differently from what is needed.)*

Fortunately there is an easy workaround for this issue. We will simply get a stream of **int** values and use the (byte) cast operator to discard the 24 most significant bits from each **int** value. It is probably safe to assume that if the **int** values are random, the eight least significant bits of each **int** value will also be random. Given that assumption, Listing 11 gets and populates the **melody** array with those values and then returns a reference to the **melody** array.

The sound of a stream of random numbers

When you listen to the WhiteNoise audio file, you hear the result of transforming those random numbers into electrical currents that circulate through the computer speakers. That transformation is accomplished by the call to the **playOrFileData** method as the last statement in the constructor for the **MusicComposer04** class in Listing 4 .

Random numbers don't produce a very pleasant sound. Future modules will populate the melody array with values that are somewhat more interesting than random **byte** values.

Listing 11 also signals the end of the **getMelody** method and the end of the **WhiteNoise** class.

Run the program

I encourage you to copy the code from Listing 12 through Listing 16 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2010-Your First Sound Program
- File: Jbs2010.htm
- Published: 08/26/14
- Revised: 09/29/15

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the class files required by this program are provided below.

Listing 12 . The class named AudioPlayOrFile01.

```
/*File AudioPlayOrFile01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
*****************************************************************************/
import javax.sound.sampled.*;
import java.io.*;
import java.util.*;

public class AudioPlayOrFile01{
  //An object of this class is used to either play the sound in the array
  // named melody or to write it into an audio file of type AU.

  //The following are general instance variables used to create a
  // SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

  AudioFormatParameters01 audioParams;
  byte[] melody;
  String playOrFile;//"play" to play immediately or a fileName to write
                    // an output file of type AU.
  //--------------------------------------------------------------------//

  public AudioPlayOrFile01(AudioFormatParameters01 audioParams,
                           byte[] melody,
                           String playOrFile){//constructor

    this.audioParams = audioParams;
    this.melody = melody;
    this.playOrFile = playOrFile;
  }//end constructor
  //--------------------------------------------------------------------//
```

Listing 12 . The class named AudioPlayOrFile01.

```
  //This method plays or files the synthetic audio data that has been genera
  // and saved in an array.
  void playOrFileData() {
    try{
      //Get an input stream on the byte array containing the data
      InputStream byteArrayInputStream = new ByteArrayInputStream(melody);

      //Get the required audio format
      audioFormat = new AudioFormat(audioParams.sampleRate,
                                    audioParams.sampleSizeInBits,
                                    audioParams.channels,
                                    audioParams.signed,
                                    audioParams.bigEndian);

      //Get an audio input stream from the ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                                     byteArrayInputStream,
                                     audioFormat,
                                     melody.length/audioFormat.getFrameSize(

      //Get info on the required data line
      DataLine.Info dataLineInfo = new DataLine.Info(SourceDataLine.class,
                                                     audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)AudioSystem.getLine(dataLineInfo);

      //Decide whether to play the audio data immediately, or to write it
      // into an audio file of type AU based on the incoming parameter named
      // playOrFile.
      if(playOrFile.toUpperCase().equals("PLAY")){
        //Create a thread to play back the data and start it running.  It wi
        // run until all the data has been played back
        new PlayAudioThread().start();
      }else{
        //Write the data to an output file with the name provided by the
        // incoming parameter named playOrFile.
        try{
          AudioSystem.write(audioInputStream,
                            AudioFileFormat.Type.AU,
                            new File(playOrFile + ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//=============================================================================
```

Listing 12 . The class named AudioPlayOrFile01.

```
  //Inner class to play back the data that was saved.
  class PlayAudioThread extends Thread{
    //This is a working buffer used to transfer the data between the
    // AudioInputStream and the SourceDataLine.  The size is rather arbitrar
    byte playBuffer[] = new byte[16384];

    public void run(){
      try{
        //Open and start the SourceDataLine
        sourceDataLine.open(audioFormat);
        sourceDataLine.start();

        int cnt;
        //Get beginning of elapsed time for playback
        long startTime = new Date().getTime();

        //Transfer the audio data to the speakers
        while((cnt = audioInputStream.read(
                                    playBuffer,0,playBuffer.length)) != -
          //Keep looping until the input read method returns -1 for empty
          // stream.
          if(cnt > 0){
            //Write data to the internal buffer of the data line where it wi
            // be delivered to the speakers in real time
            sourceDataLine.write(playBuffer, 0, cnt);
          }//end if
        }//end while

        //Block and wait for internal buffer of the SourceDataLine to become
        // empty.
        sourceDataLine.drain();


        //Get and display the elapsed time for the previous playback.
        int elapsedTime = (int)(new Date().getTime() - startTime);
        System.out.println("Elapsed time: " + elapsedTime);

        //Finish with the SourceDataLine
        sourceDataLine.stop();
        sourceDataLine.close();
      }catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }//end catch

    }//end run
  }//end inner class PlayAudioThread
  //===================================================================
}//end AudioPlayOrFile01 class
```

..

Listing 13 . The class named AudioFormatParameters01.

```
/*File AudioFormatParameters01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************

public class AudioFormatParameters01{
  //The following are audio format parameters used by the Java audio system.
  // They may be modified by the signal generator at runtime.  Values allowe
  // by Java SDK 1.4.1 are shown in comments.
  public float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100 samples per second
  public int sampleSizeInBits = 16;
  //Allowable 8,16
  public int channels = 1;
  //Allowable 1 for mono and 2 for stereo
  public boolean signed = true;
  //Allowable true,false
  public boolean bigEndian = true;
  //Allowable true,false
}//end class AudioFormatParameters01
//==========================================================================
```

..

Listing 14 . The class named MusicComposer04.

```
/*File MusicComposer04.java
Copyright 2014, R.G.Baldwin
Revised 08/20/14

This program creates and plays three seconds of monaural white or pink noise
It works in conjunction with the following classes:

WhiteNoise
AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
```

Listing 14 . The class named MusicComposer04.

```
/**************************************************************************

public class MusicComposer04{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //--------------------------------------------------------------------

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the string w
  // be used as a filename for an audio file of type AU. In the latter case,
  // it must be a string that would be valid as a file name for the operatin
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer04(args);
  }//end main
  //--------------------------------------------------------------------

  public MusicComposer04(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if

    //Get a populated array containing audio data for white or pink noise.
    WhiteNoise whiteNoise = new WhiteNoise(audioParams,this.args,melody);
    melody = whiteNoise.getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //--------------------------------------------------------------------
}//end class MusicComposer04.java
//====================================================================
```

--

Listing 15 . The class named AudioSignalGenerator02.

```
/*File AudioSignalGenerator02.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is an abstract class that serves as the base class for several other
classes that can be used to create melodies of different types.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This clsss can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //--------------------------------------------------------------------

  //Constructor
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------

  //The following abstract method must be overridden in a subclass for this
  // class to be of any value.
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
//====================================================================
```

..

Listing 16 . The class named WhiteNoise.

```
/*File WhiteNoise.java
```

Listing 16 . The class named WhiteNoise.

```
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is a simple class that can be used to create "white noise"
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class WhiteNoise extends AudioSignalGenerator02{

  public WhiteNoise(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-------------------------------------------------------------------------


  //This method returns an array containing three seconds of monaural
  // white noise.

  byte[] getMelody(){
    //Set the audio parameters to mono
    audioParams.channels = 1;//superfluous -- default value
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    Random generator = new Random(new Date().getTime());
    for(int cnt = 0;cnt < melody.length; cnt++){
      melody[cnt] = (byte)generator.nextInt();
    }//end for loop
    return melody;
  }//end method getMelody
  //-------------------------------------------------------------------------
```

Listing 16 . The class named WhiteNoise.

```
}//end class WhiteNoise
//===========================================================================
```

-end-

Jbs2020-Square Wave Sound
This module explains how to write a program that creates an audio output consisting of a 1000 Hz square wave in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains how to write a program that creates an audio output consisting of a 1000 Hz square wave.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

General background information

What is sound ?

An earlier module titled *Jbs2000-What is Sound?* explained some of the physical characteristics of sound. It also explained that sound can be produced when a computer program causes the diaphragms on audio speakers connected to the computer to move back and forth or vibrate.

White noise

An earlier module titled *Jbs2010-Your First Sound Program* explained how a random number generator can be used in a program to cause the speaker diaphragms to move back and forth or vibrate in a random fashion producing what is commonly called white noise or pink noise. *(Click WhiteNoise to hear a sample of white or pink noise. You should be able to play the audio file with any standard media player that can handle the AU file type. In case you are on the OpenStax site and you are unable to download the audio file, click the* ***Legacy Site*** *link at the top of this page to switch over to the same module on the Legacy site.You should be able to download the audio file from there.)*

White noise is totally unorganized or random. Beginning with this module, we will start applying more organization to the sound until we reach the point where we can write a program that simulates a player piano. *(Click Greensleeves to hear a simulation of a player piano.)*

A square wave

In this module, we will write a program that attempts to cause the diaphragms to move back and forth instantaneously switching from one position to another by creating audio data based on a square wave. *(Click SquareWave to hear a sample of a 1000 Hz square wave. A future module will compare the sound of a 1000 Hz square wave with the sound of a pure 1000 Hz sinusoidal tone.)*

An audio graph of a square wave

I explained the concept of an *audio graph* in the earlier module titled *Jbs2000-What is Sound?* but I will explain it again here for your convenience. *(Click AudioGraphSquareWave to hear an audio representation of the graph of a square wave.)*

This audio graph file contains an 8-second melody consisting of 32 uniformly spaced pulses at different frequencies. The frequencies *(pitches)* of the pulses are centered on middle-C *(261.63 Hz)* . The frequency deviation from middle-C versus time is based on a square wave function with a frequency of 0.5 Hz.

Points on graph

Each pulse represents one point on a graph of the square wave. Pulses with frequencies at or above middle-C are delivered to the left speaker. Pulses with frequencies below middle-C are delivered to the right speaker.

The audio output can be thought of as an audio representation of a graph of a square wave. Pulses with frequencies above middle-C represent points on the positive lobe of the square wave. Increasing pitch represents increasing amplitude on the graph of the square wave.

Pulses with frequencies below middle-C can be thought of as representing points on the negative lobe of the square wave. In this case, decreasing pitch represents points on the square wave that are further from the horizontal axis in the negative direction.

Pulses with a frequency of middle-C can be thought of as representing points on the horizontal axis with a value of zero but there are no points on the horizontal axis for a square wave.

Four complete cycles of the 0.5 Hz square wave are represented by the 32 pulses in the 8-second melody.

Hopefully, by listening to this audio file, you can get an idea of the shape of a square wave.

Discussion and sample code

This program requires the following five classes:

- AudioFormatParameters01 *(see Listing 7)*
- AudioPlayOrFile01 *(see Listing 8)*
- AudioSignalGenerator02 *(see Listing 9)*
- MusicComposer05 *(see Listing 10)*
- SquareWave *(see Listing 11)*

Three classes are unchanged

I won't bore you by repeating the discussion from earlier modules. The first three classes in the above list are completely unchanged from the module titled *Jbs2010-Your First Sound Program* . Therefore, I won't discuss them further in this module.

The class named MusicComposer05

The class named **MusicComposer05** differs from the previous version only in the following respects:

- Changes in the explanatory comments.
- Replacement of the term **WhiteNoise** with the term **SquareWave** .
- Replacement of the term **whiteNoise** with the term **squareWave** .

Therefore, I also won't discuss this class further in this module.

The class named SquareWave

A complete listing of the class named **SquareWave** is provided in Listing 11 . I will break this class down and explain it in fragments.

Beginning of the class named SquareWave

The sound that you heard when you listened to the audio file named SquareWave was produced by the **getMelody** method of the **SquareWave** class. The **SquareWave** class begins in Listing 1 and the **getMelody** method begins in Listing 2 .

The code in Listing 1 differs from the corresponding **WhiteNoise** code from the earlier module only with respect to the name of the class. Therefore, I won't discuss it further.

Listing 1 . Beginning of the class named SquareWave.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class SquareWave extends AudioSignalGenerator02{

  public SquareWave(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
```

Beginning of the getMelody method

This method returns an array containing three seconds of monaural audio data for a square wave at 1000 Hz.

Listing 2 shows the beginning of the overridden **getMelody** method. *(Recall that an abstract version of this method is inherited from the class named **AudioSignalGenerator02** -- see Listing 9)*

The code in Listing 2 is essentially the same as the corresponding **WhiteNoise** code from the earlier module. Therefore, I won't discuss it further in this module.

Listing 2 . Beginning of the getMelody method.

Listing 2 . Beginning of the getMelody method.

```
  byte[] getMelody(){
    //Recall that the default for channels is 1 for mono.
    System.out.println("audioParams.channels = " +
audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are
8000,11025,
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes
per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*

bytesPerSampPerChan*audioParams.channels)];
    System.out.println("melody.length = " + melody.length);
```

Required audio data format

As you learned in earlier modules, an object of the **AudioPlayOrFile01** class accepts an object of the **AudioFormatParameters01** class along with an audio data array object of type **byte[]** and a **String** object for a file name and uses that information to either play the data in the audio array immediately or write it into an audio output file of type AU.

Normally the audio data array must be formatted in a specific way as partially defined by the contents of the **AudioFormatParameters01** object. In the case of white noise, however, there is no order or organization to the bytes of audio data. Therefore, our only requirement was to ascertain that the proper number of signed random byte values were used to populate the array.

That is not the case in this module. Our audio data is organized and we must meet the required format for the audio array.

Given the values that we are using in the **AudioFormatParameters01** object, the format requirements for monaural and stereo are shown below. *(Note that in both cases, each audio value must be a signed 16-bit value decomposed into a pair of 8-bit bytes.)*

Monaural, channels = 1

For mono, each successive pair of bytes in the array must contain one audio value. The element with the lower index must contain the most significant eight bits of the 16-bit audio value.

Stereo, channels = 2

For stereo, alternating pairs of bytes must each contain one audio value in the same byte order as for mono. One pair of bytes is routed to the left speaker and the other pair of bytes is routed to the right speaker *(almost)* simultaneously.

Within the four bytes, the pair with the lowest index is routed to the left speaker and the other pair is routed to the right speaker.

Generating the square wave

Local working variables

Listing 3 declares three working variables. The variable named **val** specifies the amplitude of the square wave. You will see what the other two variable are used for later.

Listing 3 . Local working variables.

```
int val = 8000;//amplitude value of square wave
byte byteLow = 0;
byte byteHigh = 0;
```

Set the frequency of the square wave

Listing 4 shows the beginning of a **for** loop that is used to populate the audio data array. Note that this **for** loop traverses the array in steps of two bytes per iteration. This will be important later when we examine the code that populates the array.

Listing 4 . Set the frequency of the square wave.

Listing 4 . Set the frequency of the square wave.

```
    for(int cnt = 0;cnt < melody.length; cnt+=2){

      //Change this value to change the fundamental frequency.
      //8 results in 1000 Hz
      //16 results in 500 Hz
      //32 results in 250 Hz, etc.

      if(cnt % 8 == 0){
        //Change sign
        val = -val;
      }//end if
```

The amplitude of the square wave switches back and forth between **val** and **-val** on a regular basis. The value used with the modulus operator in the **if** statement in Listing 4 determines how often that switch will take place. If it switches every eight bytes at a sampling rate of 8000 samples per second, the resulting fundamental frequency will be 1000 Hz. The comments in Listing 4 indicate other possibilities.

Note: Do the math:

```
(8 bytes/switch)*(2 switches/cycle)*(1 samp/2 bytes)*
                                        (1 sec/8000 samp) = 0.001 sec/cycle
or 1000 cycles/sec or 1000 Hz
```

Decompose val into two bytes

Listing 5 decomposes the 16 least significant bits *(LSB)* of **val** into a pair of 8-bit bytes, discarding the 16 most significant bits *(MSB)* of the 32-bit **int** value.

Listing 5 . Decompose val into two bytes.

```
      byteLow = (byte)val;//discard all but 8 lsb
      byteHigh = (byte)(val >> 8);//shift right 8 and discard all but 8
lsb
```

The first statement in Listing 5 uses a (byte) cast to discard the 24 MSB of a copy of **val** and to store the remaining eight LSB in the variable named **byteLow** . This does not modify the value actually stored in the variable named **val** .

The second statement in Listing 5 begins by shifting a copy of **val** 8 bits to the right, thereby discarding the eight LSB and keeping the remaining 24 MSB of the original 32 bits. This operation extends the sign bit to the right in conjunction with the shift.

Then the second statement uses a (byte) cast to discard the 24 MSB of the same copy of **val** and to store the remaining eight LSB in the variable named **byteHigh** .

There is a more elegant, but much more cryptic way to do this, which I will show you in a future module. I'm using this approach at this time because it should be easy for you to understand what is actually happening.

Populate the audio data array

Listing 6 assigns the values of **byteHigh** and **byteLow** to the elements in the audio data array in the required order described above . Recall that the for loop iterates through the array two bytes per iteration. Thus, each iteration produces one audio sample.

Listing 6 . Populate the audio data array.

```
      melody[cnt] = byteHigh;
      melody[cnt + 1] = byteLow;

    }//end for loop

    return melody;

  }//end method getMelody
  //-------------------------------------------------------------------
-----//

}//end class SquareWave
```

Listing 6 also signals the end of the **for** loop and returns the populated array's reference after the loop terminates.

Finally, Listing 6 signals the end of the **getMelody** method and the end of the **SquareWave** class.

Run the program

I encourage you to copy the code from Listing 7 through Listing 11 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2020-Square Wave Sound
- File: Jbs2020.htm
- Published: 08/27/14
- Revised: 09/29/15

Complete program listings

Source code for the classes required by this program is provided below.

Listing 7 . The class named AudioFormatParameters01.

Listing 7 . The class named AudioFormatParameters01.

```
/*File AudioFormatParameters01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************

public class AudioFormatParameters01{
  //The following are audio format parameters used by the Java audio system.
  // They may be modified by the signal generator at runtime.  Values allowe
  // by Java SDK 1.4.1 are shown in comments.
  public float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100
  public int sampleSizeInBits = 16;
  //Allowable 8,16
  public int channels = 1;
  //Allowable 1,2
  public boolean signed = true;
  //Allowable true,false
  public boolean bigEndian = true;
  //Allowable true,false
}//end class AudioFormatParameters01
//============================================================================
```

..

Listing 8 . The class named AudioPlayOrFile01.

```
/*File AudioPlayOrFile01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************
import javax.sound.sampled.*;
import java.io.*;
import java.util.*;

public class AudioPlayOrFile01{
  //An object of this class is used to either play the sound in the array
  // named melody or to write it into an audio file of type AU.

  //The following are general instance variables used to create a
  // SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;
```

Listing 8 . The class named AudioPlayOrFile01.

```
  AudioFormatParameters01 audioParams;
  byte[] melody;
  String playOrFile;//"play" to play immediately or a fileName to write
                    // an output file of type AU.
  //-----------------------------------------------------------------------

  public AudioPlayOrFile01(AudioFormatParameters01 audioParams,
                           byte[] melody,
                           String playOrFile){//constructor

    this.audioParams = audioParams;
    this.melody = melody;
    this.playOrFile = playOrFile;
  }//end constructor
  //-----------------------------------------------------------------------

  //This method plays or files the synthetic audio data that has been genera
  // and saved in an array.
  void playOrFileData() {
    try{
      //Get an input stream on the byte array containing the data
      InputStream byteArrayInputStream = new ByteArrayInputStream(melody);

      //Get the required audio format
      audioFormat = new AudioFormat(audioParams.sampleRate,
                                    audioParams.sampleSizeInBits,
                                    audioParams.channels,
                                    audioParams.signed,
                                    audioParams.bigEndian);

      //Get an audio input stream from the ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                                     byteArrayInputStream,
                                     audioFormat,
                                     melody.length/audioFormat.getFrameSize(

      //Get info on the required data line
      DataLine.Info dataLineInfo = new DataLine.Info(SourceDataLine.class,
                                                     audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)AudioSystem.getLine(dataLineInfo);

      //Decide whether to play the audio data immediately, or to write it
      // into an audio file of type AU based on the incoming parameter named
      // playOrFile.
      if(playOrFile.toUpperCase().equals("PLAY")){
        //Create a thread to play back the data and start it running.  It wi
        // run until all the data has been played back
        new PlayAudioThread().start();
      }else{
        //Write the data to an output file with the name provided by the
        // incoming parameter named playOrFile.
        try{
```

Listing 8 . The class named AudioPlayOrFile01.

```
          AudioSystem.write(audioInputStream,
                            AudioFileFormat.Type.AU,
                            new File(playOrFile + ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//===========================================================================

  //Inner class to play back the data that was saved.
  class PlayAudioThread extends Thread{
    //This is a working buffer used to transfer the data between the
    // AudioInputStream and the SourceDataLine.  The size is rather arbitrar
    byte playBuffer[] = new byte[16384];

    public void run(){
      try{
        //Open and start the SourceDataLine
        sourceDataLine.open(audioFormat);
        sourceDataLine.start();

        int cnt;
        //Get beginning of elapsed time for playback
        long startTime = new Date().getTime();

        //Transfer the audio data to the speakers
        while((cnt = audioInputStream.read(
                                      playBuffer,0,playBuffer.length)) != -
          //Keep looping until the input read method returns -1 for empty
          // stream.
          if(cnt > 0){
            //Write data to the internal buffer of the data line where it wi
            // be delivered to the speakers in real time
            sourceDataLine.write(playBuffer, 0, cnt);
          }//end if
        }//end while

        //Block and wait for internal buffer of the SourceDataLine to become
        // empty.
        sourceDataLine.drain();


        //Get and display the elapsed time for the previous playback.
        int elapsedTime = (int)(new Date().getTime() - startTime);
        System.out.println("Elapsed time: " + elapsedTime);

        //Finish with the SourceDataLine
        sourceDataLine.stop();
```

Listing 8 . The class named AudioPlayOrFile01.

```
        sourceDataLine.close();
      }catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }//end catch

    }//end run
  }//end inner class PlayAudioThread
  //===================================================================
}//end AudioPlayOrFile01 class
```

..

Listing 9 . The class named AudioSignalGenerator02.

Listing 9 . The class named AudioSignalGenerator02.

```
/*File AudioSignalGenerator02.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is an abstract class that serves as the base class for several other
classes that can be used to create melodies of different types.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This class can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //--------------------------------------------------------------------//

  //Constructor
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------//

  //The following abstract method must be overridden in a subclass for this
  // class to be of any value.
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
//======================================================================//
```

..

Listing 10 . The class named MusicComposer05.

```
/*File MusicComposer05.java
Copyright 2014, R.G.Baldwin
```

Listing 10 . The class named MusicComposer05.

```
Revised 08/22/14

This program works in conjunction with the following classes to create and p
three seconds of monaural audio based on a square wave at 1000 cycles per
second.

SquareWave
AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
**************************************************************************/

public class MusicComposer05{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //----------------------------------------------------------------------

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the string w
  // be used as a filename for an audio file of type AU. In the latter case,
  // it must be a string that would be valid as a file name for the operatin
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer05(args);
  }//end main
  //----------------------------------------------------------------------

  public MusicComposer05(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if

    //Get a populated array containing audio data for the square wave.
    SquareWave squareWave = new SquareWave(audioParams,this.args,melody);
```

Listing 10 . The class named MusicComposer05.

```
    melody = squareWave.getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-------------------------------------------------------------------------
}//end class MusicComposer05.java
//=========================================================================
```

..

Listing 11 . The class named SquareWave.

```
/*File SquareWave.java
Copyright 2014, R.G.Baldwin
Revised 08/22/14

This class can be used to create a square wave with 1000 cycles per second
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class SquareWave extends AudioSignalGenerator02{

  public SquareWave(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-------------------------------------------------------------------------


  //This method returns an array containing three seconds of a square wave
  // at 1000 cycles per second.

  byte[] getMelody(){
    //Recall that the default for channels is 1 for mono.
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
```

Listing 11 . The class named SquareWave.

```
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                  bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    int val = 8000;//amplitude value of square wave
    byte byteLow = 0;
    byte byteHigh = 0;
    for(int cnt = 0;cnt < melody.length; cnt+=2){

      if(cnt % 8 == 0){
        //Change sign
        val = -val;
      }//end if

      //Create two bytes that contain a 16-bit representation of the value.
      byteLow = (byte)val;//discard all but 8 lsb
      byteHigh = (byte)(val >> 8);//shift right 8 and discard all but 8 lsb

      //Deposit the bytes into the array
      melody[cnt] = byteHigh;
      melody[cnt + 1] = byteLow;

    }//end for loop
    return melody;
  }//end method getMelody
  //-----------------------------------------------------------------------

}//end class SquareWave
//=========================================================================
```

-end-

Jbs2030-A Pure Sinusoidal Tone
This module explains how to write a program that creates an audio output consisting of a pure sinusoidal tone at 1000 Hz in a format that is accessible to blind students. It also explains how to use the ByteBuffer class to populate an array of bytes with data of type short.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains how to write a program that creates an audio output consisting of a pure sinusoidal tone at 1000 Hz in a format that is accessible to blind students. It also explains how to use the **ByteBuffer** class to populate an array of bytes with data of type **short** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

General background information

A square wave

In an earlier module, we wrote a program that causes the diaphragms on computer speakers to move back and forth by creating audio data based on a square wave. Click SquareWave to hear a sample of a 1000 Hz square wave. *(You should be able to play the audio file with any standard media player that can handle the AU file type. In case you are on the OpenStax site and you are unable to download the audio file, click the **Legacy Site** link at the top of this page to switch over to the same module on the Legacy site. You should be able to download the audio file from there.)*

An audio graph of a square wave

I explained the concept of an *audio graph* in earlier modules. Click AudioGraphSquareWave to hear an audio representation of the graph of a square wave.

A pure sinusoidal tone

In this module, we will write a program that causes the diaphragms on computer speakers to move back and forth by creating audio data based on a sinusoidal function. Click ToneMono to hear a sample of a 1000 Hz sinusoidal function. Compare this sound with the sound of a SquareWave with the same fundamental frequency.

The difference explained

Depending on the quality of your speakers and the sensitivity of your ears, you may be able to hear a hint of higher frequency components in the square wave sound. A sinusoid represents a single frequency. A square wave with the same fundamental frequency contains a component with the same frequency as the sinusoid. However, it also contains some higher-frequency harmonics at lower intensity levels. In other words, the spectrum of a square wave is not a single frequency. Instead, it has a large peak at the fundamental frequency plus smaller peaks at higher frequencies that are harmonically related to the fundamental frequency.

An audio graph of a sinusoid

Click AudioGraphSinusoidal to hear an audio representation of the graph of a sinusoid. Compare this with AudioGraphSquareWave , which is an audio representation of the graph of a square wave.

I will explain the code that I wrote to create these audio graphs in a future module. Once you understand that code, you should be able to write the code to create audio graphs of many different mathematical functions, such as sine, cosine, tangent, cotangent, secant, cosecant, parabola, hyperbola, exponentials, half circle, half ellipse, and many others. *(See functions.wolfram.com for thousands of formulas of hundreds of functions.)*

Discussion and sample code

This program requires the following five classes:

- AudioFormatParameters01 *(see Listing 5)*
- AudioPlayOrFile01 *(see Listing 6)*
- AudioSignalGenerator02 *(see Listing 7)*
- MusicComposer06 *(see Listing 8)*
- ToneMono *(see Listing 9)*

Three classes are unchanged

I won't bore you by repeating the discussion from earlier modules. The first three classes in the above list are completely unchanged from the module titled *Jbs2010-Your First Sound Program* . Therefore, I won't discuss them further in this module.

The class named MusicComposer06

The class named **MusicComposer06** differs from the previous version only in the following respects:

- Changes in the explanatory comments.
- Replacement of the term **WhiteNoise** with the term **ToneMono** .
- Replacement of the term **whiteNoise** with the term **toneMono** .

Therefore, I also won't discuss this class further in this module.

The class named ToneMono

A complete listing of the class named **ToneMono** is provided in Listing 9 . I will break this class down and explain it in fragments.

Beginning of the class named ToneMono

The sound that you heard when you listened to the audio file named ToneMono was produced by the **getMelody** method of the **ToneMono** class. The **ToneMono** class begins in Listing 1 and the **getMelody** method begins in Listing 2 .

The code in Listing 1 differs from the corresponding **WhiteNoise** code from the earlier module only with respect to the name of the class. Therefore, I won't discuss it further.

Listing 1 . Beginning of the class named ToneMono.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class ToneMono extends AudioSignalGenerator02{

  public ToneMono(AudioFormatParameters01 audioParams,
                  String[] args,
                  byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
```

Beginning of the getMelody method

This method returns an array containing three-seconds of a pure sinusoidal tone. When this array is processed, a 1000 Hz tone is emitted with equal amplitude from the left and right speakers. It is interesting to compare this sound with the sound of a square wave with the same fundamental frequency.

Listing 2 shows the beginning of the overridden **getMelody** method. *(Recall that an abstract version of this method is inherited from the class named **AudioSignalGenerator02** -- see Listing 7)*

With one exception, the code in Listing 2 is essentially the same as the corresponding **WhiteNoise** code from the earlier module. The exception is the statement that declares a variable named **freq** and sets its value to 1000.0. As you will see later, the value stored in this variable establishes the frequency of the sinusoidal tone. Beyond that, I won't discuss the code in Listing 2 any further.

Listing 2 . Beginning of the getMelody method.

Listing 2 . Beginning of the getMelody method.

```
  byte[] getMelody(){
    //Recall that the default is channels=1 for monaural.
    System.out.println("audioParams.channels = " +
audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are
8000,11025,
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Set the frequency of the tone.
    double freq = 1000.0;

    //Create an output data array sufficient to contain the tone
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes
per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*

bytesPerSampPerChan*audioParams.channels)];
    System.out.println("melody.length = " + melody.length);
```

Required audio data format

I explained the required format of the audio data in the **melody** array in an earlier module. I will repeat that explanation here for convenience.

Given the values that we are using in the **AudioFormatParameters01** object, the format requirements for monaural and stereo are shown below. *(Note that in both cases, each audio value must be a signed 16-bit value decomposed into a pair of 8-bit bytes.)*

Monaural, channels = 1

For mono, each successive pair of bytes in the array must contain one audio value. The element with the lower index must contain the most significant eight bits of the 16-bit audio value.

Stereo, channels = 2

For stereo, alternating pairs of bytes must each contain one audio value in the same byte order as for mono. One pair of bytes is routed to the left speaker and the other pair of bytes is routed to the right speaker *(almost)* simultaneously.

Within the four bytes, the pair with the lowest index is routed to the left speaker and the other pair is routed to the right speaker.

I will also remind you that the code in the **SquareWave** class used bit shifting and casting to decompose the **short** value into a pair of **byte** values. We will accomplish that in a different and somewhat simpler way in this module.

Decomposing a short into two bytes using ByteBuffer

Recall that the abstract class named **AudioSignalGenerator02** *(see Listing 7)* declares an instance variable of type **ByteBuffer** named **byteBuffer** . This variable is inherited into the **ToneMono** class and is available to the **getMelody** method.

I encourage you to go to the Java Standard Edition documentation and read about the class named **ByteBuffer** . This is a very powerful class with a lot of capability.

In this module, we will use an object of the **ByteBuffer** class to eliminate some of the complexity involved in decomposing a **short** audio value into a pair of **byte** values and depositing the two **byte** values in the **melody** array.

Instantiate and prepare a ByteBuffer object

Using terminology from the Java SE documentation, the statement in Listing 3 creates a **ByteBuffer** object and saves that object's reference in the inherited variable named **byteBuffer** by *wrapping an existing **byte** array (**melody**) into a buffer* .

Listing 3 . Instantiate and prepare a ByteBuffer object.

```
byteBuffer = ByteBuffer.wrap(melody);
```

Why use a ByteBuffer object ?

Normally the elements of a **byte** array can only be accessed using indexed square bracket *([index])* notation. Wrapping the array in a **ByteBuffer** object makes it possible to access the elements in the array by calling methods that are defined in the **ByteBuffer** class. For example, the array can be populated from beginning to end using successive calls to the overloaded **put** methods of the **ByteBuffer** class without regard for the actual index of the elements *(provided that you don't try to exceed the length of the array)* .

Although it isn't obvious, this solves the problem of decomposing the **short** value into a pair of **byte** values and depositing them in the correct order in the melody **array** . One of the methods of the **ByteBuffer** class is named **putShort** . As you will see later, that method makes it possible to "put" a **short** value into our **ByteBuffer** object *(and hence into our **melody** array)* with a single statement. The **putShort** method will automatically decompose the **short** value into two **byte** values and deposit them in the proper order in the array.

Compute and deposit audio samples in melody array

Listing 4 begins by computing the melody length in samples as the product of the length in seconds and the sampling rate in samples per second.

Listing 4 . Compute and deposit audio samples in melody array.

```
    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Compute the audio sample values and deposit them in the output
array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      //Deposit audio data for both channels in mono.
      byteBuffer.putShort((short)(8000*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //-----------------------------------------------------------------------
-----//

}//end class ToneMono
```

Then Listing 4 enters a **for** loop for the purpose of

- Computing the time in seconds for the current sample.
- Computing the value of the **Math.sin** function for that time.
- Scaling that value up by 8000 to attain a reasonable audio level.
- Casting that result to type **short** .
- Using the **putShort** method of the **ByteBuffer** class discussed earlier to deposit the **short** value in the next two bytes in the **melody** array in a single statement.

If you go back and examine the code for this part of the **SquareWave** program, you will see that the use of the **putShort** method of the **ByteBuffer** class has simplified things considerably.

Finally the code in Listing 4

- Returns a reference to the **melody** array so that it can be processed by the code in the object of the **MusicComposer06** class.
- Signals the end of the **getMelody** method of the **ToneMono** class.
- Signals the end of the **ToneMono** class.

Run the program

I encourage you to copy the code from Listing 5 through Listing 9 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2030-A Pure Sinusoidal Tone
- File: Jbs2030.htm
- Published: 08/27/14
- Revised: 09/29/15

Complete program listings

Complete listings of the classes discussed in this module are provided below.

Listing 5 . The class named AudioFormatParameters01.

Listing 5 . The class named AudioFormatParameters01.

```
/*File AudioFormatParameters01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************

public class AudioFormatParameters01{
  //The following are audio format parameters used by the Java audio system.
  // They may be modified by the signal generator at runtime.  Values allowe
  // by Java SDK 1.4.1 are shown in comments.
  public float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100 samples per second
  public int sampleSizeInBits = 16;
  //Allowable 8,16
  public int channels = 1;
  //Allowable 1 for mono and 2 for stereo
  public boolean signed = true;
  //Allowable true,false
  public boolean bigEndian = true;
  //Allowable true,false
}//end class AudioFormatParameters01
//==========================================================================
```

..

Listing 6 . The class named AudioPlayOrFile01.

```
/*File AudioPlayOrFile01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************
import javax.sound.sampled.*;
import java.io.*;
import java.util.*;

public class AudioPlayOrFile01{
  //An object of this class is used to either play the sound in the array
  // named melody or to write it into an audio file of type AU.

  //The following are general instance variables used to create a
  // SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;
```

Listing 6 . The class named AudioPlayOrFile01.

```
  AudioFormatParameters01 audioParams;
  byte[] melody;
  String playOrFile;//"play" to play immediately or a fileName to write
                    // an output file of type AU.
  //-----------------------------------------------------------------------

  public AudioPlayOrFile01(AudioFormatParameters01 audioParams,
                           byte[] melody,
                           String playOrFile){//constructor

    this.audioParams = audioParams;
    this.melody = melody;
    this.playOrFile = playOrFile;
  }//end constructor
  //-----------------------------------------------------------------------

  //This method plays or files the synthetic audio data that has been genera
  // and saved in an array.
  void playOrFileData() {
    try{
      //Get an input stream on the byte array containing the data
      InputStream byteArrayInputStream = new ByteArrayInputStream(melody);

      //Get the required audio format
      audioFormat = new AudioFormat(audioParams.sampleRate,
                                    audioParams.sampleSizeInBits,
                                    audioParams.channels,
                                    audioParams.signed,
                                    audioParams.bigEndian);

      //Get an audio input stream from the ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                                     byteArrayInputStream,
                                     audioFormat,
                                     melody.length/audioFormat.getFrameSize(

      //Get info on the required data line
      DataLine.Info dataLineInfo = new DataLine.Info(SourceDataLine.class,
                                                     audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)AudioSystem.getLine(dataLineInfo);

      //Decide whether to play the audio data immediately, or to write it
      // into an audio file of type AU based on the incoming parameter named
      // playOrFile.
      if(playOrFile.toUpperCase().equals("PLAY")){
        //Create a thread to play back the data and start it running.  It wi
        // run until all the data has been played back
        new PlayAudioThread().start();
      }else{
        //Write the data to an output file with the name provided by the
        // incoming parameter named playOrFile.
        try{
```

Listing 6 . The class named AudioPlayOrFile01.

```
          AudioSystem.write(audioInputStream,
                            AudioFileFormat.Type.AU,
                            new File(playOrFile + ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//===========================================================================

  //Inner class to play back the data that was saved.
  class PlayAudioThread extends Thread{
    //This is a working buffer used to transfer the data between the
    // AudioInputStream and the SourceDataLine.  The size is rather arbitrar
    byte playBuffer[] = new byte[16384];

    public void run(){
      try{
        //Open and start the SourceDataLine
        sourceDataLine.open(audioFormat);
        sourceDataLine.start();

        int cnt;
        //Get beginning of elapsed time for playback
        long startTime = new Date().getTime();

        //Transfer the audio data to the speakers
        while((cnt = audioInputStream.read(
                                      playBuffer,0,playBuffer.length)) != -
          //Keep looping until the input read method returns -1 for empty
          // stream.
          if(cnt > 0){
            //Write data to the internal buffer of the data line where it wi
            // be delivered to the speakers in real time
            sourceDataLine.write(playBuffer, 0, cnt);
          }//end if
        }//end while

        //Block and wait for internal buffer of the SourceDataLine to become
        // empty.
        sourceDataLine.drain();


        //Get and display the elapsed time for the previous playback.
        int elapsedTime = (int)(new Date().getTime() - startTime);
        System.out.println("Elapsed time: " + elapsedTime);

        //Finish with the SourceDataLine
        sourceDataLine.stop();
```

Listing 6 . The class named AudioPlayOrFile01.

```
        sourceDataLine.close();
      }catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }//end catch

    }//end run
  }//end inner class PlayAudioThread
  //===================================================================
}//end AudioPlayOrFile01 class
```

..

Listing 7 . The class named AudioSignalGenerator02.

Listing 7 . The class named AudioSignalGenerator02.

```
/*File AudioSignalGenerator02.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is an abstract class that serves as the base class for several other
classes that can be used to create melodies of different types.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This class can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //--------------------------------------------------------------------//

  //Constructor
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------//

  //The following abstract method must be overridden in a subclass for this
  // class to be of any value.
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
//======================================================================//
```

..

Listing 8 . The class named MusicComposer06.

```
/*File MusicComposer06.java
Copyright 2014, R.G.Baldwin
Revised 08/22/14
```

Listing 8 . The class named MusicComposer06.

```
This program works in conjunction with the following classes to create and p
three seconds of monaural audio based on a pure sinusoidal function at 1000
cycles per second.

ToneMono
AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
**************************************************************************

public class MusicComposer06{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //----------------------------------------------------------------------

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the string w
  // be used as a filename for an audio file of type AU. In the latter case,
  // it must be a string that would be valid as a file name for the operatin
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer06(args);
  }//end main
  //----------------------------------------------------------------------

  public MusicComposer06(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if

    //Get a populated array containing audio data for the pure sinusoidal to
    ToneMono toneMono = new ToneMono(audioParams,this.args,melody);
    melody = toneMono.getMelody();
```

Listing 8 . The class named MusicComposer06.

```
    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-------------------------------------------------------------------------
}//end class MusicComposer06.java
//=========================================================================
```

..

Listing 9 . The class named ToneMono.

```
/*File ToneMono.java
Copyright 2014, R.G.Baldwin
Revised 08/22/14

This class that can be used to create a melody consisting of a single pure
sinusoidal tone at 1000 Hz.

The class introduces the use of ByteBuffer.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public class ToneMono extends AudioSignalGenerator02{

  public ToneMono(AudioFormatParameters01 audioParams,
                  String[] args,
                  byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-------------------------------------------------------------------------


  //This method generates a three-second pure sinusoidal tone. A 1000 Hz tor
  // is emitted with equal amplitude from the left and right speakers. It is
  // interesting to compare this sound with the sound of a square wave with
  // the same fundamental frequency.

  byte[] getMelody(){
    //Recall that the default is channels=1 for monaural.
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
```

Listing 9 . The class named ToneMono.

```
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Set the frequency of the tone.
    double freq = 1000.0;

    //Create an output data array sufficient to contain the tone
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                  bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Compute the audio sample values and deposit them in the output array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      //Deposit audio data for both channels in mono.
      byteBuffer.putShort((short)(8000*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //---------------------------------------------------------------------------

}//end class ToneMono
//===========================================================================
```

-end-

Jbs2040-An Audio Graph of a Sinusoid
This module explains how to create an audio graph of a sinusoid in a format that is accessible to blind students. The program could easily be modified to create audio graphs of other functions such as parabolas, exponentials, etc. The module also explains how to create stereo sound.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains how to create an audio graph of a sinusoid in a format that is accessible to blind students. The program could easily be modified to create audio graphs of other functions such as parabolas, exponentials, etc.

Previous modules have created only monaural sound. This module also explains how to create stereo sound.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

General background information

Previous modules have introduced you to the concept of an audio graph consisting of audio pulses that represent points on the graph of a function. Click AudioGraphSquareWave to hear an audio graph of a square wave function. Click AudioGraphSinusoidal to hear an audio graph of a sinusoid function. *(You should be able to play these audio files with any standard media player that can handle the AU file type. In case you are on the OpenStax site and you are unable to download the audio files, click the* ***Legacy Site*** *link at the top of this page to switch over to the same module on the Legacy site. You should be able to download the audio files from there.)*

In this module, I will explain how to create the audio graph of the sinusoid. The program could easily be modified to create audio graphs of other functions such as parabolas, exponentials, etc.

Previous modules have created only monaural sound. This module also explains how to create stereo sound.

Discussion and sample code

This program requires the following five classes:

- AudioFormatParameters01
- AudioPlayOrFile01
- AudioSignalGenerator02
- MusicComposer07 *(see Listing 9)*
- *AudioGraphSinusoidal (see Listing 10)*

Three classes are unchanged

I won't bore you by repeating the discussion from earlier modules. The first three classes in the above list are completely unchanged from the module titled *Jbs2010-Your First Sound Program* . By now, you probably have source code files and compiled class files for those three classes. If not, you can obtain the source code from the earlier module titled *Jbs2010-Your First Sound Program* . Because they haven't changed, I won't discuss these classes further in this module.

The class named MusicComposer07

The class named **MusicComposer07** differs from the previous version only in the following respects:

- Changes in the explanatory comments.
- Replacement of the term **WhiteNoise** with the term **AudioGraphSinusoidal** .
- Replacement of the term **whiteNoise** with the term **audioGraphSinusoidal** .

Therefore, I also won't discuss this class further in this module.

The class named AudioGraphSinusoidal

A complete listing of the class named **AudioGraphSinusoidal** is provided in Listing 10 . I will break this class down and explain it in fragments.

Beginning of the class named AudioGraphSinusoidal

The sound that you heard when you listened to the audio file named AudioGraphSinusoidal was produced by the **getMelody** method of the **AudioGraphSinusoidal** class. The **AudioGraphSinusoidal** class begins in Listing 1 and the **getMelody** method begins in Listing 2 .

The code in Listing 1 differs from the corresponding **WhiteNoise** code from the earlier module only with respect to the name of the class. Therefore, I won't discuss it further.

Listing 1 . Beginning of the class named AudioGraphSinusoidal.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraphSinusoidal extends AudioSignalGenerator02{

  public AudioGraphSinusoidal(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
```

Beginning of the getMelody method

This method returns a melody array that will play an 8-second melody consisting of 32 pulses at different frequencies. The frequencies of the pulses are centered on middle-C (261.63 Hz). The frequency deviation from middle-C versus time is based on a sinusoidal function with a frequency of 0.5 Hz.

Each pulse represents one point on a graph of the sinusoid. Pulses with frequencies at or above middle-C are delivered to the left speaker. Pulses with frequencies below middle-C are delivered to the right speaker.

The audio output can be thought of as an audio representation of a graph of a sinusoid. Pulses with frequencies above middle-C represent points on the positive lobe of the sinusoid. Increasing pitch represents increasing amplitude on the graph of the sinusoid. Pulses with frequencies below middle-C can be thought of as representing points on the negative lobe of the sinusoid. In this case, decreasing pitch represents points on the sinusoid that are

further from the horizontal axis in the negative direction. Pulses with a frequency of middle-C can be thought of as representing points on the horizontal axis with a value of zero.

In order to eliminate pops and clicks caused by abrupt frequency changes in the audio signal, the amplitude of each pulse is scaled by a triangular *(rooftop)* function that has a value that is zero at both ends and 1.0 in the center with a linear progression from the center to the ends in both directions.

Four complete cycles of the 0.5 Hz sinusoid are represented by the 32 pulses in the 8-second melody.

Listing 2 shows the beginning of the overridden **getMelody** method. *(Recall that an abstract version of this method is inherited from the class named **AudioSignalGenerator02** .*

Listing 2 . Beginning of the getMelody method.

```
  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " +
audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample
rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 8.0;

    //Set the center frequency. Audio pulses will be generated above and
    // below this frequency to represent points on the graph of a
    // sinusoidal function.
    double centerFreq = 261.63;//middle C

    //Create an output array of sufficient size to contain the melody at
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*

bytesPerSampPerChan*audioParams.channels)];
    System.out.println("melody.length = " + melody.length);
```

Note the statement in Listing 2 that sets the value for **channels** to 2 for stereo. That is new in this module.

Also note the statement that sets the center frequency to 261.63 Hz. That is the pitch that represents zero amplitude in the audio graph of the sinusoidal function.

Otherwise, there is nothing new in Listing 2 .

Controlling the output volume for each speaker

Recall that I explained earlier that pulses that represent points on the positive lobe of the sinusoidal function are delivered to the left speaker. Pulses that represent points on the negative lobe of the sinusoidal function are delivered to the right speaker.

This is accomplished by scaling the audio values emitted by each speaker by a factor that is either zero or non zero. When scaled by zero, there is no audio output from a speaker. This makes it possible to switch the audio back and forth between the two speakers.

This is accomplished later using the variables that are declared in Listing 3 .

Listing 3 . Declare speaker volume variables.

```
double gain = 0.0;
double leftGain = 0.0;
double rightGain = 0.0;
```

Miscellaneous operations

Listing 4 continues with several miscellaneous operations. The embedded comments should be sufficient to explain these operations.

Listing 4 . Miscellaneous operations.

Listing 4 . Miscellaneous operations.

```
    //Declare a variable that is used to control the frequency of each
pulse.
    double freq = 0.0;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Set the length of each pulse in seconds and in samples.
    double pulseLengthInSec = 0.25;//in seconds
    int pulseLengthInSamples = (int)
(pulseLengthInSec*audioParams.sampleRate);
```

Map the sinusoidal function values into pulse frequencies

Listing 5 begins with a **for** loop that eventually deposits audio output values in the **melody** array.

Listing 5 . Map the sinusoidal function values into pulse frequencies.

Listing 5 . Map the sinusoidal function values into pulse frequencies.

```
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      if(cnt%pulseLengthInSamples == 0){
        //Time to create a new pulse at a different pitch. Compute the
        // frequency for the next pulse to represent a point on a
sinusoidal
        // function of time. This section of code could easily be
modified
        // to create audio graphs of many different functions.

        //Evaluate and scale the function
        double val = 0.35 * Math.sin(2*Math.PI*0.5*time);

        //Compute the frequency for the next pulse as a deviation from
the
        // center frequency. For this scaled sinusoidal function, the
Range
        // is from 0.65*centerFreq to 1.35*centerFreq or from 170.05 Hz
        // to 353.2 Hz.
        freq = (1+val)*centerFreq;

      }//end if
```

Compute the current time

The code inside the **for** loop in Listing 5 begins by computing the time in seconds for the current sample. This value will be used later to evaluate the sinusoidal function that is being graphed using audio. It will also be used to compute the values of the higher-frequency sinusoids used to control the pitch of the pulses.

Compute the pitch of the pulse

An **if** statement is used with the modulus operator to cause the pitch to change from one pulse to the next. The code inside the **if** statement computes the amplitude of the sinusoidal function being graphed at the current point in time. This value will range from -1.0 to +1.0 for a sinusoidal function, but the range may be different if you apply this approach to other functions such as a parabola. It then uses that amplitude value to compute the required pitch of the pulse, at, above, or below the center frequency to represent the amplitude of the sinusoidal function at, above, or below 0.0. The new pitch value is stored in the variable named **freq** .

Shape the pulse amplitude from beginning to end

When I first wrote this program, I noticed there were a lot of pops and clicks as the pulse frequency changed abruptly from one pulse to the next. To eliminate that problem, I applied a shaping scale factor to the amplitude of each pulse to cause it to:

- Start at zero at the beginning of the pulse.

- Increase linearly to a maximum value at the center of the pulse.
- Decrease linearly back to zero at the end of the pulse.

This is accomplished using the gain factor that is computed in Listing 6 . The gain factor ranges from 0.0 at the ends to 1.0 in the center of the pulse. I will leave it as an exercise for the student to decipher this code. *(Hint: It would help to know the equation for a straight line.)*

Listing 6 . Shape the pulse amplitude from beginning to end.

```
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;

      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -

cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      }//end if

      //Set the gain to a value that is compatible with 16-bit audio
data.
      gain = 8000*gain;
```

Listing 6 ends by scaling the gain factor by 8000 to produce a reasonable audio output level.

Compute the gain for the left and right speakers

Listing 7 examines the sign *(positive or negative)* of the current value of the sinusoidal function *(based on the pulse frequency relative to the center frequency)* and uses that information to compute gain factors that will turn one speaker on and turn the other speaker off.

Listing 7 . Compute the gain for the left and right speakers.

Listing 7 . Compute the gain for the left and right speakers.

```
if(freq >= centerFreq){
  leftGain = gain;
  rightGain = 0;//switch off the right channel
}else{
  rightGain = gain;
  leftGain = 0;//switch off the left channel
}//
```

Required audio data format

As you learned in an earlier module, given the values that we are using in the **AudioFormatParameters01** object, the format requirements for monaural and stereo are shown below. *(Note that in both cases, each audio value must be a signed 16-bit value decomposed into a pair of 8-bit bytes.)*

Monaural, channels = 1

For mono, each successive pair of bytes in the **melody** array must contain one audio value. The element with the lower index must contain the most significant eight bits of the 16-bit audio value.

Stereo, channels = 2

For stereo, alternating pairs of bytes must each contain one audio value in the same byte order as for mono. One pair of bytes is routed to the left speaker and the other pair of bytes is routed to the right speaker *(almost)* simultaneously.

Within the four bytes, the pair with the lowest index is routed to the left speaker and the other pair is routed to the right speaker.

You learned how to use the **putShort** method belonging to an object of the **ByteBuffer** class to deposit the **short** data into the **byte** array in the earlier module titled *Jbs2030-A Pure Sinusoidal Tone* .

Deposit stereo audio data in the melody file

Listing 8 used the information from above to compute the current audio value of the pulse and to deposit it into two consecutive pairs of bytes in the **melody** array. Note that the **putShort** method is called twice, once for each channel. Note also that the values of **leftGain** and **rightGain** are used to scale each audio value so that it will be emitted from only one of the two stereo speakers.

Listing 8 . Deposit stereo audio data in the melody file.

Listing 8 . Deposit stereo audio data in the melody file.

```
      byteBuffer.putShort((short)
(leftGain*Math.sin(2*Math.PI*freq*time)));
      byteBuffer.putShort((short)
(rightGain*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //----------------------------------------------------------------------
-----//

}//end class AudioGraphSinusoidal
```

Listing 8 returns a reference to the **melody** array when the **for** loop terminates.

Listing 8 also signals the end of the **getMelody** method and the end of the **AudioGraphSinusoidal** class.

Controlling stereo speaker output levels

This module demonstrates one way to control the output levels from the two speakers in a stereo melody. It is not necessary that one speaker be turned completely *off* and the other speaker turned completely *on* as is the case here. The relative levels of the audio outputs from the two speakers can be controlled by adjusting the relative values of the left and right gain values.

Run the program

I encourage you to copy the code from Listing 9 and Listing 10 . Retrieve the necessary source code for the other three classes from the module titled *Jbs2010-Your First Sound Program* . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2040-An Audio Graph of a Sinusoid
- File: Jbs2040.htm
- Published: 08/27/14
- Revised: 09/29/15

Note: Disclaimers:

Financial : Although the **OpenStax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF. You also need to know that Prof. Baldwin receives no financial compensation from **OpenStax CNX** even if you purchase the PDF version of the collection.
In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receives compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **OpenStax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.
Affiliation :: Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of two of the classes discussed in this module are provided below. Source code for the remaining three classes can be obtained from the module titled *Jbs2010-Your First Sound Program* .

Listing 9 . The class named MusicComposer07.

```
/*File MusicComposer07.java
Copyright 2014, R.G.Baldwin
Revised 08/23/14

This program creates and plays an audio graph of a sinusoid. It works in
conjunction with the following classes:

AudioGraphSinusoidal
AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
**********************************************************************/

public class MusicComposer07{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;
```

Listing 9 . The class named MusicComposer07.

```
  //A place to store the incoming args array.
  String[] args;
  //----------------------------------------------------------------------

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the string w
  // be used as a filename for an audio file of type AU. In the latter case,
  // it must be a string that would be valid as a file name for the operatir
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer07(args);
  }//end main
  //----------------------------------------------------------------------

  public MusicComposer07(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if

    //Get a populated array containing audio data for white or pink noise.
    AudioGraphSinusoidal audioGraphSinusoidal =
                         new AudioGraphSinusoidal(audioParams,this.args,melod
    melody = audioGraphSinusoidal.getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //----------------------------------------------------------------------
}//end class MusicComposer07.java
```

..

Listing 10 . The class named AudioGraphSinusoidal.

```
/*File AudioGraphSinusoidal.java
Copyright 2014, R.G.Baldwin
Revised 08/23/14

This class can be used to create an 8-second melody consisting of 32 pulses
```

Listing 10 . The class named AudioGraphSinusoidal.

```
at different frequencies.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraphSinusoidal extends AudioSignalGenerator02{

  public AudioGraphSinusoidal(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-------------------------------------------------------------------


  //This method returns a melody array that will play an 8-second melody
  // consisting of 32 pulses at different frequencies. The frequencies of th
  // pulses are centered on middle-C (261.63 Hz).

  //The frequency deviation from middle-C versus time is based on a sinusoid
  // function with a frequency of 0.5 Hz. Each pulse represents one point on
  // a graph of the sinusoid. Pulses with frequencies at or above middle-C a
  // delivered to the left speaker. Pulses with frequencies below middle-C a
  // delivered to the right speaker.

  //The audio output can be thought of as an audio representation of a graph
  // of a sinusoid. Pulses with frequencies above middle-C represent points
  // on the positive lobe of the sinusoid. Increasing pitch represents
  // increasing amplitude on the graph of the sinusoid. Pulses with
  // frequencies below middle-C can be thought of as representing points on
  // the negative lobe of the sinusoid. In this case, decreasing pitch
  // represents points on the sinusoid that are further from the horizontal
  // axis in the negative direction. Pulses with a frequency of middle-C can
  // be thought of as representing points on the horizontal axis with a valu
  // of zero.

  //In order to eliminate pops and clicks caused by abrupt frequency changes
  // in the audio signal, the amplitude of each pulse is scaled by a
  // triangular (rooftop) function that has a value that is zero at both end
  // and 1.0 in the center with a linear progression from the center to the
  // ends in both directions.

  //Four complete cycles of the 0.5 Hz sinusoid are represented by the 32
  // pulses in the 8-second melody.

  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;
```

Listing 10 . The class named AudioGraphSinusoidal.

```
    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 8.0;

    //Set the center frequency. Audio pulses will be generated above and
    // below this frequency to represent points on the graph of a
    // sinusoidal function.
    double centerFreq = 261.63;//middle C

    //Create an output array of sufficient size to contain the melody at
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Declare variables used to control the output volume on the left and
    // right speaker channels. These values will be used to cause pulses
    // representing negative values of the sinusoidal function to emit from
    // one speaker and pulses representing positive values to emit from
    // the other speaker.
    double gain = 0.0;
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare a variable that is used to control the frequency of each pulse
    double freq = 0.0;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Set the length of each pulse in seconds and in samples.
    double pulseLengthInSec = 0.25;//in seconds
    int pulseLengthInSamples = (int)(pulseLengthInSec*audioParams.sampleRate

    //Compute the audio sample values and deposit them in the output melody
    // array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      if(cnt%pulseLengthInSamples == 0){
        //Time to create a new pulse at a different pitch. Compute the
        // frequency for the next pulse to represent a point on a sinusoidal
        // function of time. This section of code could easily be modified
        // to create audio graphs of many different functions.
```

Listing 10 . The class named AudioGraphSinusoidal.

```
        //Evaluate and scale the function
        double val = 0.35 * Math.sin(2*Math.PI*0.5*time);

        //Compute the frequency for the next pulse as a deviation from the
        // center frequency. For this scaled sinusoidal function, the Range
        // is from 0.65*centerFreq to 1.35*centerFreq or from 170.05 Hz
        // to 353.2 Hz.
        freq = (1+val)*centerFreq;

      }//end if

      //Deposit audio data in the melody for each channel. Scale the amplitu
      // of each pulse with a triangular scale factor (rooftop shape) to
      // minimize the undesirable pops and clicks that occur when there is a
      // abrupt change in the frequency from one pulse to the next. The
      // following gain factor ranges from 0.0 at the ends to 1.0 in the
      // center of the pulse.
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -
                        cnt%pulseLengthInSamples)/(double)pulseLengthInSampl
      }//end if

      //Set the final gain to a value that is compatible with 16-bit audio
      // data.
      gain = 8000*gain;

      //Switch the left and right channels on and off depending on the locat
      // of the pulse frequency relative to the center frequency.
      if(freq >= centerFreq){
        leftGain = gain;
        rightGain = 0;//switch off the right channel
      }else{
        rightGain = gain;
        leftGain = 0;//switch off the left channel
      }//

      //Compute scaled pulse values and deposit them into the melody.
      byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time)));
      byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //-----------------------------------------------------------------------

}//end class AudioGraphSinusoidal
```

-end-

Jbs2050-Runtime Polymorphism with Java Sound
This module demonstrates runtime polymorphism using an array of an abstract type populated with references to objects of eight different subclasses of that abstract type in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It demonstrates runtime polymorphism using an array of the abstract type **AudioSignalGenerator02** populated with references to objects of eight different subclasses of **AudioSignalGenerator02** in a format that is accessible to blind students.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

- Listing 11 . The class named ToneMono.
- Listing 12 . The class named TonesStereo.
- Listing 13 . The class named WhiteNoise.
- Listing 14 . The class named AudioFormatParameters01.
- Listing 15 . The class named AudioPlayOrFile01.
- Listing 16 . The class named AudioSignalGenerator02.

General background information

We have reached a plateau of sorts in our study of Java sound. This and previous modules have produced hard-coded sounds. Future modules will be more flexible. For example, one future module will simulate a player piano. It will accept information about a melody as input data at runtime and will tailor the sound based on that information. For example, in future modules, you will learn to write a program to produce sounds like Greensleeves and MaryHadALittlelamb .

But I am getting ahead of myself. In previous modules, you have learned:

- A little about the physical characteristics of sound.
- How to produce white or pink noise.
- How to produce square wave sound.
- How to produce a pure sinusoidal tone.
- How to produce an audio graph of a sinusoid.

In this module, you will learn how to produce a few more types of sounds:

- An audio graph for a square wave.
- A frequency-modulated sweep in stereo.
- A stereo ping pong effect.
- A pair of pure tones of different frequencies in stereo.

More importantly, you will learn how to bring all of this knowledge together to demonstrate runtime polymorphism using sound.

This program requires access to the following classes:

- AudioGraphSinusoidal
- AudioGraphSquareWave
- FMSweep
- MusicComposer08
- SquareWave
- StereoPingpong
- ToneMono
- TonesStereo
- WhiteNoise
- AudioFormatParameters01
- AudioPlayOrFile01
- AudioSignalGenerator02

You will find source code for all of these classes in Listing 5 through Listing 16 .

You are already familiar with some of these classes because they were used in earlier modules.

The driver class for this program is the class named **MusicComposer08** . That is the only one of the twelve classes that I will discuss in detail. I will leave it as an exercise for the students at this point to study and understand the remaining eleven classes.

Discussion and sample code

The class named MusicComposer08

The purpose of this program is to demonstrate *late binding* and *runtime polymorphism* . This is accomplished by using an array of the abstract type **AudioSignalGenerator02** populated with references to objects of the **following eight classes** :

- AudioGraphSinusoidal
- AudioGraphSquareWave
- FMSweep
- SquareWave
- StereoPingpong
- ToneMono
- TonesStereo
- WhiteNoise

Each of the classes listed above extends the abstract class named **AudioSignalGenerator02** and overrides the inherited abstract method named **getMelody** .

After the array is populated, a random number generator is used to get a random index into the array. The **getMelody** method is called on the reference pointed to by the random index. This causes the sound created by that particular audio object to be played or filed.

This is an example of late binding and runtime polymorphism because the compiler cannot possibly know the value of the random index when the program is compiled.

As in previous modules, the sound can be played immediately or can be saved in an audio file of type AU for playback later. You should be able to play the audio file with any standard media player that can handle the AU file type.

The program also requires access to the following classes:

- AudioFormatParameters01
- AudioPlayOrFile01
- AudioSignalGenerator02

Beginning of the class named MusicComposer08

Listing 1 shows the beginning of the class named **MusicComposer08** . The code in Listing 1 is essentially the same as code that I have explained in several previous modules. Therefore, no further discussion of the code in Listing 1 will be provided in this module.

Listing 1 . Beginning of the class named MusicComposer08.

Listing 1 . Beginning of the class named MusicComposer08.

```
import java.util.Random;

public class MusicComposer08{
  //Instantiate an object containing audio format parameters with
predefined
  // values. They may be modified by the signal generator at runtime.
Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class
definition.
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //---------------------------------------------------------------------
-----//

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the
string will
  // be used as a filename for an audio file of type AU. In the latter
case,
  // it must be a string that would be valid as a file name for the
operating
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer08(args);
  }//end main
  //---------------------------------------------------------------------
-----//

  public MusicComposer08(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the
command line.
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if
```

Create and populate an array holding references to audio objects

Listing 2 begins by creating an array capable of holding references to eight audio objects instantiated from subclasses of the **AudioSignalGenerator02** class. Note that the type of each element in this array is

AudioSignalGenerator02 .

Then Listing 2 populates the array elements with references to eight different audio objects instantiated from those classes. Even though an object may be instantiated from the class named **FMSweep** , for example, that object's reference is being stored as type **AudioSignalGenerator02** .

Listing 2 . Create and populate an array holding references to audio objects.

```
    AudioSignalGenerator02[] audioObjects = new
AudioSignalGenerator02[8];

    audioObjects[0] = new
AudioGraphSinusoidal(audioParams,this.args,melody);
    audioObjects[1] = new
AudioGraphSquareWave(audioParams,this.args,melody);
    audioObjects[2] = new FMSweep(audioParams,this.args,melody);
    audioObjects[3] = new SquareWave(audioParams,this.args,melody);
    audioObjects[4] = new StereoPingpong(audioParams,this.args,melody);
    audioObjects[5] = new ToneMono(audioParams,this.args,melody);
    audioObjects[6] = new TonesStereo(audioParams,this.args,melody);
    audioObjects[7] = new WhiteNoise(audioParams,this.args,melody);
```

Override an inherited abstract method

The abstract class named **AudioSignalGenerator02** declares an abstract method named **getMelody** . Each subclass of the abstract class must define a concrete version of *(override)* the inherited abstract method or the subclass itself be declared abstract.

All eight of the subclasses listed in Listing 2 override the **getMelody** method. The behavior of each overridden version of the method is significantly different from one class to the next.

When the **getMelody** method is called on a reference to a subclass object that is stored as the superclass type, the resulting behavior is that of the subclass method and not that of the superclass method. That is runtime polymorphism.

Call the getMelody method on a subclass object selected at random

Listing 3 selects a reference to an audio object at random and calls the **getMelody** method on the object. As in previous modules, the reference to the **byte** array returned by the method is saved in the **byte[]** variable named **melody** . Unlike before, however, at this point, we don't know what melody was saved there. We only know that it will be one of eight possible melodies and we won't know which one until we actually play the melody. *(At least that would be true if the code in Listing 3 didn't access and print the type of the object pointed to by the random index.)*

Listing 3 . Call the getMelody method on a subclass object selected at random.

```
    Random randomGenerator = new Random();
    int randomIndex = randomGenerator.nextInt(8);
    System.out.println("randomIndex = " + randomIndex);
    System.out.println(audioObjects[randomIndex].getClass());

    melody = audioObjects[randomIndex].getMelody();
```

Some new melodies

The following melodies are new to this module:

- AudioGraphSquareWave
- FMSweep
- StereoPingpong
- TonesStereo

You will find listings of the source code used to create these melodies later in this module.

Play or file the melody

The code in Listing 4 uses the standard procedure to either play or file the **melody** that was created in Listing 3 .

Listing 4 . Play or file the melody.

```
    new
AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();

  }//end constructor
  //-------------------------------------------------------------------
-----//
}//end class MusicComposer08.java
```

An extremely important concept

While this may seem like a rather innocuous result, runtime polymorphism is an extremely important concept in object-oriented programming. One of the important applications of runtime polymorphism is the *Java Collections Framework* . A series of accessible modules that explain the framework begins with *Java4010 : Getting Started with Java Collections* .

Run the program

I encourage you to copy the code from Listing 5 through Listing 16 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2050-Runtime Polymorphism with Java Sound
- File: Jbs2050.htm
- Published: 08/27/14
- Revised: 09/29/15

Complete program listings

Complete listing of the twelve classes required by the program discussed in this module are provided below.

Listing 5 . The class named AudioGraphSinusoidal.

```
/*File AudioGraphSinusoidal.java
Copyright 2014, R.G.Baldwin
Revised 08/23/14

This class can be used to create an 8-second melody consisting of 32 pulses
```

Listing 5 . The class named AudioGraphSinusoidal.

```
at different frequencies.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraphSinusoidal extends AudioSignalGenerator02{

  public AudioGraphSinusoidal(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-----------------------------------------------------------------------


  //This method returns a melody array that will play an 8-second melody
  // consisting of 32 pulses at different frequencies. The frequencies of th
  // pulses are centered on middle-C (261.63 Hz).

  //The frequency deviation from middle-C versus time is based on a sinusoid
  // function with a frequency of 0.5 Hz. Each pulse represents one point on
  // a graph of the sinusoid. Pulses with frequencies at or above middle-C a
  // delivered to the left speaker. Pulses with frequencies below middle-C a
  // delivered to the right speaker.

  //The audio output can be thought of as an audio representation of a graph
  // of a sinusoid. Pulses with frequencies above middle-C represent points
  // on the positive lobe of the sinusoid. Increasing pitch represents
  // increasing amplitude on the graph of the sinusoid. Pulses with
  // frequencies below middle-C can be thought of as representing points on
  // the negative lobe of the sinusoid. In this case, decreasing pitch
  // represents points on the sinusoid that are further from the horizontal
  // axis in the negative direction. Pulses with a frequency of middle-C can
  // be thought of as representing points on the horizontal axis with a valu
  // of zero.

  //In order to eliminate pops and clicks caused by abrupt frequency changes
  // in the audio signal, the amplitude of each pulse is scaled by a
  // triangular (rooftop) function that has a value that is zero at both end
  // and 1.0 in the center with a linear progression from the center to the
  // ends in both directions.

  //Four complete cycles of the 0.5 Hz sinusoid are represented by the 32
  // pulses in the 8-second melody.

  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;
```

Listing 5 . The class named AudioGraphSinusoidal.

```
    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 8.0;

    //Set the center frequency. Audio pulses will be generated above and
    // below this frequency to represent points on the graph of a
    // sinusoidal function.
    double centerFreq = 261.63;//middle C

    //Create an output array of sufficient size to contain the melody at
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Declare variables used to control the output volume on the left and
    // right speaker channels. These values will be used to cause pulses
    // representing negative values of the sinusoidal function to emit from
    // one speaker and pulses representing positive values to emit from
    // the other speaker.
    double gain = 0.0;
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare a variable that is used to control the frequency of each pulse
    double freq = 0.0;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Set the length of each pulse in seconds and in samples.
    double pulseLengthInSec = 0.25;//in seconds
    int pulseLengthInSamples = (int)(pulseLengthInSec*audioParams.sampleRate

    //Compute the audio sample values and deposit them in the output melody
    // array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      if(cnt%pulseLengthInSamples == 0){
        //Time to create a new pulse at a different pitch. Compute the
        // frequency for the next pulse to represent a point on a sinusoidal
        // function of time. This section of code could easily be modified
        // to create audio graphs of many different functions.
```

Listing 5 . The class named AudioGraphSinusoidal.

```
        //Evaluate and scale the function
        double val = 0.35 * Math.sin(2*Math.PI*0.5*time);

        //Compute the frequency for the next pulse as a deviation from the
        // center frequency. For this scaled sinusoidal function, the Range
        // is from 0.65*centerFreq to 1.35*centerFreq or from 170.05 Hz
        // to 353.2 Hz.
        freq = (1+val)*centerFreq;

      }//end if

      //Deposit audio data in the melody for each channel. Scale the amplitu
      // of each pulse with a triangular scale factor (rooftop shape) to
      // minimize the undesirable pops and clicks that occur when there is a
      // abrupt change in the frequency from one pulse to the next. The
      // following gain factor ranges from 0.0 at the ends to 1.0 in the
      // center of the pulse.
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -
                        cnt%pulseLengthInSamples)/(double)pulseLengthInSampl
      }//end if

      //Set the final gain to a value that is compatible with 16-bit audio
      // data.
      gain = 8000*gain;

      //Switch the left and right channels on and off depending on the locat
      // of the pulse frequency relative to the center frequency.
      if(freq >= centerFreq){
        leftGain = gain;
        rightGain = 0;//switch off the right channel
      }else{
        rightGain = gain;
        leftGain = 0;//switch off the left channel
      }//

      //Compute scaled pulse values and deposit them into the melody.
      byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time)));
      byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //---------------------------------------------------------------------

}//end class AudioGraphSinusoidal
//=====================================================================
```

.....

Listing 6 . The class named AudioGraphSquareWave.

```
/*File AudioGraphSquareWave.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This class can be used to create an 8-second melody consisting of 32 pulses
at different frequencies.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraphSquareWave extends AudioSignalGenerator02{

  public AudioGraphSquareWave(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-------------------------------------------------------------------


  //This method returns a melody array that will play an 8-second melody
  // consisting of 32 pulses at two different frequencies. The two frequenci
  // are equidistant on either side of middle-C.

  //The frequency deviation from middle-C versus time is based on a square w
  // function with a frequency of 0.5 Hz. That is, the function
  // switches between equal positive and negative values once per second
  // producing a periodic function where each cycle is two seconds
  // long.

  //Each pulse represents one point on
  // a graph of the square wave. Pulses with frequencies at or above
  // middle-C are delivered to the left speaker. Pulses with
  // frequencies below middle-C are delivered to the right speaker.
  // Middle-C is considered to represent the horizontal axis of the
  // graph with a value of zero.

  //The audio output can be thought of as an audio representation of a graph
  // of a square wave. Pulses with frequencies above middle-C represent poir
  // on the positive lobe of the square wave. Increasing pitch represents
  // increasing amplitude on the graph of the square wave. Pulses with
  // frequencies below middle-C can be thought of as representing points on
  // the negative lobe of the square wave. In this case, decreasing pitch
  // represents points on the square wave that are further from the horizont
  // axis in the negative direction.

  //In order to eliminate pops and clicks caused by abrupt frequency changes
  // in the audio signal, the amplitude of each pulse is scaled by a
```

Listing 6 . The class named AudioGraphSquareWave.

```
  // triangular (rooftop) function that has a value that is zero at both enc
  // and 1.0 in the center with a linear progression from the center to the
  // ends in both directions.

  //Four complete cycles of the 0.5 Hz square wave are represented by the 32
  // pulses in the 8-second melody.

  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 8.0;

    //Set the center frequency.
    double centerFreq = 261.63;//middle C

    //Create an output array of sufficient size to contain the melody at
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Declare variables used to control the output volume on the left and
    // right speaker channels.
    double gain = 0.0;
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare the initial frequency deviation from center. To represent the
    // square wave, pulses will be played that are plus and minus this
    // amount relative to the center frequency.
    double deviation = 75;

    //Specify the number of pulses in one-half cycle of the square wave.
    int halfCycle = 4;

    //Declare a variable that is used to control the frequency of each pulse
    double freq = centerFreq + deviation;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);
```

Listing 6 . The class named AudioGraphSquareWave.

```
    //Set the length of each pulse in seconds and in samples.
    double pulseLengthInSec = 0.25;//in seconds
    int pulseLengthInSamples = (int)(pulseLengthInSec*audioParams.sampleRate

    //Compute the audio sample values and deposit them in the output melody
    // array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      if(cnt % (halfCycle * pulseLengthInSamples) == 0){
        //Switch to the other side of the center frequency.
        deviation = -deviation;
        freq = centerFreq + deviation;
      }//end if

      //Deposit audio data in the melody for each channel. Scale the amplitu
      // of each pulse with a triangular scale factor (rooftop shape) to
      // minimize the undesirable pops and clicks that occur when there is a
      // abrupt change in the frequency from one pulse to the next. The
      // following gain factor ranges from 0.0 at the ends to 1.0 in the
      // center of the pulse.
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -
                        cnt%pulseLengthInSamples)/(double)pulseLengthInSampl
      }//end if

      //Set the final gain to a value that is compatible with 16-bit audio
      // data.
      gain = 8000*gain;

      //Switch the left and right channels on and off depending on the locat
      // of the pulse frequency relative to the center frequency.
      if(freq >= centerFreq){
        leftGain = gain;
        rightGain = 0;//switch off the right channel
      }else{
        rightGain = gain;
        leftGain = 0;//switch off the left channel
      }//

      //Compute scaled pulse values and deposit them into the melody.
      byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time)));
      byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //-----------------------------------------------------------------------
```

Listing 6 . The class named AudioGraphSquareWave.

```
}//end class AudioGraphSquareWave
//=============================================================================
```

.....

Listing 7 . The class named FMSweep.

```
/*File FMSweep.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This class can be used to create an 8-second stereo melody consisting of a
linear frequency sweep with a linear pan between speakers.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public class FMSweep extends AudioSignalGenerator02{

  public FMSweep(AudioFormatParameters01 audioParams,
                 String[] args,
                 byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //---------------------------------------------------------------------//

  //This method returns an array that will play an 8-second melody consistir
  // of a linear frequency-modulated sweep from a low frequency of middle-C
  // to a high frequency that is one octave above middle-C. During the
  // frequency sweep, the sound pans linearly from the left speaker to the
  // right speaker.
  //
  byte[] getMelody(){

    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
```

Listing 7 . The class named FMSweep.

```
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 8.0;

    //Set the low and high frequencies to cause the sweep to cover one full
    // octave.
    double lowFreq = 261.63;//middle C
    double highFreq = 2*lowFreq;

    //Create an output data array of sufficient size to contain the melody a
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Set the overall gain to a value that is compatible with 16-bit audio
    // data.
    double gain = 8000.0;

    //Declare variables used to control the output volume on the left and
    // right speaker channels.
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare a variable that is used to control the frequency.
    double freq = 0.0;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Compute the audio sample values and deposit them in the output melody
    // array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      //Compute the frequency for this iteration
      freq = lowFreq + (highFreq - lowFreq)*cnt/sampLength;

      //Adjust the left and right gain values to cause the sound to pan
      // linearly from the left speaker to the right speaker.
      rightGain = gain*1.0*cnt/sampLength;
      leftGain = gain*1.0-rightGain;

      //Compute scaled values and deposit them into the melody.
      byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time)));
      byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)));

    }//end for loop
```

Listing 7 . The class named FMSweep.

```
    return melody;
  }//end method getMelody
  //-------------------------------------------------------------------------

}//end class FMSweep
//=========================================================================
```

.....

Listing 8 . The class named MusicComposer08.

```
/*File MusicComposer08.java
Copyright 2014, R.G.Baldwin
Revised 08/24/14

This program demonstrates late binding and runtime polymorphism using an arr
of the abstract type AudioSignalGenerator02 populated with references to
objects of the following classes:

AudioGraphSinusoidal
AudioGraphSquareWave
FMSweep
SquareWave
StereoPingpong
ToneMono
TonesStereo
WhiteNoise

Each of the classes listed above extends the abstract class named
AudioSignalGenerator02 and overrides the abstract method named getMelody.

After the array is populated, a random number generator is used to get a
random index into the array. The getMelody method is called on the reference
pointed to by the random index. This causes the sound created by that audio
object to be played or filed. This is an example of late binding because the
compiler cannot possibly know the value of the random index when the program
is compiled.

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type.

The program also requires access to the following classes:
AudioFormatParameters01
```

Listing 8 . The class named MusicComposer08.

```
AudioPlayOrFile01
AudioSignalGenerator02

Tested using JDK 1.8 under Win 7.
**************************************************************************
import java.util.Random;

public class MusicComposer08{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //--------------------------------------------------------------------

  //Command-line parameter (only one parameter is needed)
  //If "play", the sound will be played immediately. Otherwise, the string w
  // be used as a filename for an audio file of type AU. In the latter case,
  // it must be a string that would be valid as a file name for the operatin
  // system in use.
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer08(args);
  }//end main
  //--------------------------------------------------------------------

  public MusicComposer08(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[1];
      this.args[0] = "play";//Play the melody immediately
    }//end if

    //Create an array capable of holding references to eight audio objects
    // instantiated from subclasses of the AudioSignalGenerator02 class.
    AudioSignalGenerator02[] audioObjects = new AudioSignalGenerator02[8];

    //Populate the array elements with references to eight different
    // audio objects.
    audioObjects[0] = new AudioGraphSinusoidal(audioParams,this.args,melody)
    audioObjects[1] = new AudioGraphSquareWave(audioParams,this.args,melody)
    audioObjects[2] = new FMSweep(audioParams,this.args,melody);
    audioObjects[3] = new SquareWave(audioParams,this.args,melody);
    audioObjects[4] = new StereoPingpong(audioParams,this.args,melody);
    audioObjects[5] = new ToneMono(audioParams,this.args,melody);
    audioObjects[6] = new TonesStereo(audioParams,this.args,melody);
```

Listing 8 . The class named MusicComposer08.

```
    audioObjects[7] = new WhiteNoise(audioParams,this.args,melody);

    //Select an audio object at random and call the getMelody method on
    // the object.
    Random randomGenerator = new Random();
    int randomIndex = randomGenerator.nextInt(8);
    System.out.println("randomIndex = " + randomIndex);
    System.out.println(audioObjects[randomIndex].getClass());

    melody = audioObjects[randomIndex].getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();

  }//end constructor
  //-----------------------------------------------------------------------
}//end class MusicComposer08.java
//=======================================================================
```

.....

Listing 9 . The class named SquareWave.

```
/*File SquareWave.java
Copyright 2014, R.G.Baldwin
Revised 08/22/14

This class can be used to create a square wave with 1000 cycles per second
**********************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class SquareWave extends AudioSignalGenerator02{

  public SquareWave(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //-----------------------------------------------------------------------


  //This method returns an array containing three seconds of a square wave
```

Listing 9 . The class named SquareWave.

```
  // at 1000 cycles per second.

  byte[] getMelody(){
    //Recall that the default for channels is 1 for mono.
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                  bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    int val = 8000;//amplitude value of square wave
    byte byteLow = 0;
    byte byteHigh = 0;
    for(int cnt = 0;cnt < melody.length; cnt+=2){

      //Change this value to change the fundamental frequency.
      //8 results in 1000 Hz
      //16 results in 500 Hz
      //32 results in 250 Hz, etc.
      if(cnt % 8 == 0){
        //Change sign
        val = -val;
      }//end if

      //Create two bytes that contain a 16-bit representation of the value.
      byteLow = (byte)val;//discard all but 8 lsb
      byteHigh = (byte)(val >> 8);//shift right 8 and discard all but 8 lsb

      //Deposit the bytes into the array
      melody[cnt] = byteHigh;
      melody[cnt + 1] = byteLow;

    }//end for loop
    return melody;
  }//end method getMelody
  //---------------------------------------------------------------------

}//end class SquareWave
//=====================================================================
```

.....

Listing 10 . The class named StereoPingpong.

```
/*File StereoPingpong.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This class can be used to create an 4-second stereo melody that ping-pongs
back and forth between the left and right speakers.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public class StereoPingpong extends AudioSignalGenerator02{

  public StereoPingpong(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //--------------------------------------------------------------------//


  //This method returns an array that will play a four-second stereo sound
  // that ping-pongs back and forth between the left and right speakers.
  // The tone emitted by the left speaker is at a frequency of middle-C. The
  // tone emitted by the right speaker is twice that frequency. The sound
  // switches between speakers four times per second.
  byte[] getMelody(){

    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Specify the length of the melody in seconds.
    double lengthInSeconds = 4.0;

    //Set the frequency for the left speaker.
```

Listing 10 . The class named StereoPingpong.

```
    double freq = 261.63;//middle C

    //Create an output array of sufficient size to contain the melody at
    // "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Declare variables used to control the output volume on the left and
    // right speaker channels.
    double leftGain = 0.0;
    double rightGain = 8000.0;

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Set the length of each pulse in seconds and in samples.
    double pulseLengthInSec = 0.25;//in seconds
    int pulseLengthInSamples = (int)(pulseLengthInSec*audioParams.sampleRate

    //Compute the audio sample values and deposit them in the output melody
    // array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      if(cnt%pulseLengthInSamples == 0){
        //Swap gain values between channels
        double temp = leftGain;
        leftGain = rightGain;
        rightGain = temp;
      }//end if

      //Generate data for left speaker at middle-C
      double sinValue = Math.sin(2*Math.PI*(freq)*time);
      byteBuffer.putShort((short)(leftGain*sinValue));

      //Generate data for right speaker at twice the frequency of middle-C
      sinValue = Math.sin(2*Math.PI*(freq*2.0)*time);
      byteBuffer.putShort((short)(rightGain*sinValue));

    }//end for loop

    return melody;
  }//end method getMelody
  //--------------------------------------------------------------------------

}//end class StereoPingpong
//============================================================================
```

.....

Listing 11 . The class named ToneMono.

```
/*File ToneMono.java
Copyright 2014, R.G.Baldwin
Revised 08/22/14

This class that can be used to create a melody consisting of a single pure
sinusoidal tone at 1000 Hz.

The class introduces the use of ByteBuffer.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class ToneMono extends AudioSignalGenerator02{

  public ToneMono(AudioFormatParameters01 audioParams,
                  String[] args,
                  byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //------------------------------------------------------------------------


  //This method generates a three-second pure sinusoidal tone. A 1000 Hz ton
  // is emitted with equal amplitude from the left and right speakers. It is
  // interesting to compare this sound with the sound of a square wave with
  // the same fundamental frequency.

  byte[] getMelody(){
    //Recall that the default is channels=1 for monaural.
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;
```

Listing 11 . The class named ToneMono.

```
    //Set the frequency of the tone.
    double freq = 1000.0;

    //Create an output data array sufficient to contain the tone
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                    bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Compute the audio sample values and deposit them in the output array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      //Deposit audio data for both channels in mono.
      byteBuffer.putShort((short)(8000*Math.sin(2*Math.PI*freq*time)));

    }//end for loop

    return melody;
  }//end method getMelody
  //----------------------------------------------------------------------

}//end class ToneMono
//======================================================================
```

.....

Listing 12 . The class named TonesStereo.

```
/*File TonesStereo.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This class that can be used to create a melody consisting of two tones at
different frequencies emitted from the left and right stereo speakers.
**********************************************************************
```

Listing 12 . The class named TonesStereo.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class TonesStereo extends AudioSignalGenerator02{

  public TonesStereo(AudioFormatParameters01 audioParams,
                     String[] args,
                     byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //--------------------------------------------------------------------


  //This method generates a pair of three-second tones in stereo.

  //A 261.63 (middle-C) tone is emitted from the left speaker and a higher
  // frequency tone is emitted from the right speaker.

  byte[] getMelody(){
    //Set the audio parameters to stereo overriding the default value.
    audioParams.channels = 2;
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Set the primary tone frequency.
    double freq = 261.63;//middle C

    //Create an output data array sufficient to contain the tone
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                  bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    //Compute the number of audio samples in the melody.
    int sampLength = (int)(lengthInSeconds*audioParams.sampleRate);

    //Compute the audio sample values and deposit them in the output array.
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;
```

Listing 12 . The class named TonesStereo.

```
      //Deposit audio data for the first (left) channel
      byteBuffer.putShort((short)(8000*Math.sin(2*Math.PI*freq*time)));

      //Deposit audio data at a different frequency in the second (right)
      // channel.
      byteBuffer.putShort((short)(8000*Math.sin(2*Math.PI*freq*time*2.2)));

    }//end for loop

    return melody;
  }//end method getMelody
  //----------------------------------------------------------------------//

}//end class TonesStereo
//======================================================================//
```

.....

Listing 13 . The class named WhiteNoise.

```
/*File WhiteNoise.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is a simple class that can be used to create "white noise"
**********************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public class WhiteNoise extends AudioSignalGenerator02{

  public WhiteNoise(AudioFormatParameters01 audioParams,
                    String[] args,
                    byte[] melody){
    super(audioParams,args,melody);
  }//end constructor
  //----------------------------------------------------------------------//


  //This method returns an array containing three seconds of monaural
  // white noise.
```

Listing 13 . The class named WhiteNoise.

```
  byte[] getMelody(){
    //Set the audio parameters to mono
    audioParams.channels = 1;//superfluous -- default value
    System.out.println("audioParams.channels = " + audioParams.channels);

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sample rate. Allowable sample rates are 8000,1102
    // 16000,22050,44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    // Set the length of the melody in seconds
    double lengthInSeconds = 3.0;

    //Create an output data array sufficient to contain the melody
    // at "sampleRate" samples per second, "bytesPerSampPerChan" bytes per
    // sample per channel and "channels" channels.
    melody = new byte[(int)(lengthInSeconds*audioParams.sampleRate*
                                  bytesPerSampPerChan*audioParams.channels
    System.out.println("melody.length = " + melody.length);

    Random generator = new Random(new Date().getTime());
    for(int cnt = 0;cnt < melody.length; cnt++){
      melody[cnt] = (byte)generator.nextInt();
    }//end for loop
    return melody;
  }//end method getMelody
  //---------------------------------------------------------------------------

}//end class WhiteNoise
//===========================================================================
```

.....

Listing 14 . The class named AudioFormatParameters01.

Listing 14 . The class named AudioFormatParameters01.

```
/*File AudioFormatParameters01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************

public class AudioFormatParameters01{
  //The following are audio format parameters used by the Java audio system.
  // They may be modified by the signal generator at runtime.  Values allowe
  // by Java SDK 1.4.1 are shown in comments.
  public float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100 samples per second
  public int sampleSizeInBits = 16;
  //Allowable 8,16
  public int channels = 1;
  //Allowable 1 for mono and 2 for stereo
  public boolean signed = true;
  //Allowable true,false
  public boolean bigEndian = true;
  //Allowable true,false
}//end class AudioFormatParameters01
//==========================================================================
```

.....

Listing 15 . The class named AudioPlayOrFile01.

```
/*File AudioPlayOrFile01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**************************************************************************
import javax.sound.sampled.*;
import java.io.*;
import java.util.*;

public class AudioPlayOrFile01{
  //An object of this class is used to either play the sound in the array
  // named melody or to write it into an audio file of type AU.

  //The following are general instance variables used to create a
  // SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;
```

Listing 15 . The class named AudioPlayOrFile01.

```
  AudioFormatParameters01 audioParams;
  byte[] melody;
  String playOrFile;//"play" to play immediately or a fileName to write
                    // an output file of type AU.
  //-----------------------------------------------------------------------

  public AudioPlayOrFile01(AudioFormatParameters01 audioParams,
                           byte[] melody,
                           String playOrFile){//constructor

    this.audioParams = audioParams;
    this.melody = melody;
    this.playOrFile = playOrFile;
  }//end constructor
  //-----------------------------------------------------------------------

  //This method plays or files the synthetic audio data that has been genera
  // and saved in an array.
  void playOrFileData() {
    try{
      //Get an input stream on the byte array containing the data
      InputStream byteArrayInputStream = new ByteArrayInputStream(melody);

      //Get the required audio format
      audioFormat = new AudioFormat(audioParams.sampleRate,
                                    audioParams.sampleSizeInBits,
                                    audioParams.channels,
                                    audioParams.signed,
                                    audioParams.bigEndian);

      //Get an audio input stream from the ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                                     byteArrayInputStream,
                                     audioFormat,
                                     melody.length/audioFormat.getFrameSize(

      //Get info on the required data line
      DataLine.Info dataLineInfo = new DataLine.Info(SourceDataLine.class,
                                                     audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)AudioSystem.getLine(dataLineInfo);

      //Decide whether to play the audio data immediately, or to write it
      // into an audio file of type AU based on the incoming parameter named
      // playOrFile.
      if(playOrFile.toUpperCase().equals("PLAY")){
        //Create a thread to play back the data and start it running.  It wi
        // run until all the data has been played back
        new PlayAudioThread().start();
      }else{
        //Write the data to an output file with the name provided by the
        // incoming parameter named playOrFile.
        try{
```

Listing 15 . The class named AudioPlayOrFile01.

```
          AudioSystem.write(audioInputStream,
                            AudioFileFormat.Type.AU,
                            new File(playOrFile + ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//===========================================================================

  //Inner class to play back the data that was saved.
  class PlayAudioThread extends Thread{
    //This is a working buffer used to transfer the data between the
    // AudioInputStream and the SourceDataLine.  The size is rather arbitrar
    byte playBuffer[] = new byte[16384];

    public void run(){
      try{
        //Open and start the SourceDataLine
        sourceDataLine.open(audioFormat);
        sourceDataLine.start();

        int cnt;
        //Get beginning of elapsed time for playback
        long startTime = new Date().getTime();

        //Transfer the audio data to the speakers
        while((cnt = audioInputStream.read(
                                      playBuffer,0,playBuffer.length)) != -
          //Keep looping until the input read method returns -1 for empty
          // stream.
          if(cnt > 0){
            //Write data to the internal buffer of the data line where it wi
            // be delivered to the speakers in real time
            sourceDataLine.write(playBuffer, 0, cnt);
          }//end if
        }//end while

        //Block and wait for internal buffer of the SourceDataLine to become
        // empty.
        sourceDataLine.drain();


        //Get and display the elapsed time for the previous playback.
        int elapsedTime = (int)(new Date().getTime() - startTime);
        System.out.println("Elapsed time: " + elapsedTime);

        //Finish with the SourceDataLine
        sourceDataLine.stop();
```

Listing 15 . The class named AudioPlayOrFile01.

```
        sourceDataLine.close();
      }catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }//end catch

    }//end run
  }//end inner class PlayAudioThread
  //===================================================================
}//end AudioPlayOrFile01 class
```

.....

Listing 16 . The class named AudioSignalGenerator02.

Listing 16 . The class named AudioSignalGenerator02.

```
/*File AudioSignalGenerator02.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is an abstract class that serves as the base class for several other
classes that can be used to create melodies of different types.
**************************************************************************/

import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This class can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //--------------------------------------------------------------------//

  //Constructor
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------//

  //The following abstract method must be overridden in a subclass for this
  // class to be of any value.
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
//======================================================================//
```

-end-

Jbs2060-A Player Piano Simulator
This module explains how to write a program that simulates a player piano in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It explains how to write a program that simulates a player piano in a format that is accessible to blind students.

Here are some sample melodies produced by the program. *(You should be able to play these audio files with any standard media player that can handle the AU file type. In case you are on the OpenStax site and you are unable to download the audio files, click the **Legacy Site** link at the top of this page to switch over to the same module on the Legacy site. You should be able to download the **audio files from there** .)*

- AllNotes
- Chords
- FourScales
- Greensleeves
- MaryLambSimple
- MaryLambStereo
- Scales

Click here to download a zip file containing the text files and the Windows batch files needed to play these melodies. The zip file also contains the audio files of type AU listed above.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

- Listing 19 . Beginning of the method named makeMusic.
- Listing 20 . Process each array containing duration and note names in the ArrayList object.
- Listing 21 . Process each sample.
- Listing 22 . Process each piano key that is pressed.
- Listing 23 . Use a scale factor to shape the note.
- Listing 24 . The class named AudioFormatParameters01.
- Listing 25 . The class named AudioPlayOrFile01.
- Listing 26 . The class named AudioSignalGenerator02.
- Listing 27 . The class named MusicComposer09.
- Listing 28 . The class named PlayerPiano01.
- Listing 29 . The file named GreensleevesTreble.txt
- Listing 30 . The file named GreensleevesBass.txt
- Listing 31 . The file named Greensleeves.bat.

General background information

An previous module titled *Jbs2050-Runtime Polymorphism with Java Sound* summarized the kinds of things that you have learned in the earlier modules in the series. As indicated in that module, the program that I will discuss in this module will be significantly different from the code in the earlier modules. For example, the code in those modules produced hard-coded sound such as StereoPingpong and FMSweep .

The code in this module will allow you to control the sound based on the contents of a text file that you create. This allows a great deal of flexibility as indicated by the melodies in the above list .

A player piano simulator

In the late 19th and early 20th century, various companies manufactured pianos that played music automatically. The music was recorded on rolls of perforated paper. A pneumatic or electro-mechanical mechanism inside the piano read the holes in the paper and used that information to press the keys on the piano keyboard in such a way as to cause the piano to play the pre-recorded music. To a sighted observer, it looked as if an invisible person was playing the piano because the piano keys moved up and down with no hands on the keyboard.

These automated pianos were often referred to as *player pianos* . The program that I will explain in this module is a *player piano simulator* . You can "pre-record" a melody in text files *(as opposed to rolls of perforated paper)* and play those files using the simulator program. As you will learn later, a melody is defined in one *(or optionally two)* text files using standard musical notation where the notes have names like A, A#, B, C, C#, etc.

I am not a musician

I'm an engineer, not a musician. While I understand quite a bit about the physics of sound, I understand very little about how that sound is put together to create the pleasing sound that we call music. This module will describe a simulated musical instrument. Therefore, it will be necessary for me to explain the code using musical terminology. I'm sure that the explanation will contain many gaffs insofar as musical terminology is concerned. However, the purpose is to teach programming and not to teach music. Please bear with me and overlook my musical terminology gaffs.

Required classes

This program requires access to the following five classes **in the same folder** :

- AudioFormatParameters01

- AudioPlayOrFile01
- AudioSignalGenerator02
- MusicComposer09
- PlayerPiano01

You will find source code for all of these classes in Listing 24 through Listing 28 .

You are already familiar with the first three classes in the above list because they were used in earlier modules. Therefore, I won't discuss them in this module

The driver class for this program is the class named **MusicComposer09** . The player piano simulator is defined in the class named **PlayerPiano01** . I will explain those two classes in the next section.

Discussion and sample code

The class named MusicComposer09

This is the driver class for playing piano melodies. It requires the files in the above list to be in the same folder.

The melody to be played is defined in one and optionally two text files. The required text file defines the notes and the duration of those notes on the treble clef. The optional text file defines the notes and the durations of those notes on the bass clef. Those text files must exist in a folder named **Music** which is a child or immediate subfolder of the folder that contains the compiled program code.

The text file for the treble clef of a stereo rendition of Greensleeves is shown in Listing 29 . The text file for the bass clef of that same melody is shown in Listing 30 . I will explain the contents of those **files later. Click** here to download a zip file containing those two text files and other files as well, including Windows batch files.

The melody can be played immediately or can be saved in an audio file of type AU for playback later. Listing 31 contains a Windows batch file that does both. I will explain the contents of this batch file later. This batch file is also contained in the zip file mentioned above . You should be able to play the audio file with any standard media player that can handle the AU file type.

Beginning of the class named MusicComposer09

The beginning of the class named **MusicComposer09** is shown in Listing 1 . You have seen code like this in several previous modules, so no explanation beyond the embedded comments should be needed.

Listing 1 . Beginning of the class named MusicComposer09.

Listing 1 . Beginning of the class named MusicComposer09.

```
public class MusicComposer09{
  //Instantiate an object containing audio format parameters with
predefined
  // values. They may be modified by the signal generator at runtime.
Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class
definition.
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the data for the melody that will be played or
filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
```

The main method

The **main** method is shown in Listing 2 .

Listing 2 . The main method.

```
  public static void main(String[] args){
    /*Command-line parameters
    0: "play" to play immediately, fileName to create an AU file. Note
that
       the output filename cannot be named play.au.
    1: Beats per second.
    2: Name of file containing treble clef data (required).
    3: Name of file containing bass clef data (optional).
    */

    //Instantiate a new object of this class.
    new MusicComposer09(args);
  }//end main
```

The only thing that is new in Listing 2 is the description of the command-line parameters. The only thing new there is the second parameter referred to as "Beats per second." As you will see when we get into the code, the value of this parameter determines how fast the melody will be played. Low values cause the melody to be played slowly. Higher values cause the melody to be played more rapidly.

Beginning of the constructor for the MusicComposer09 class

The beginning of the constructor for the **MusicComposer09** class is shown in Listing 3 .

Listing 3 . Beginning of the constructor for the MusicComposer09 class.

```
public MusicComposer09(String[] args){//constructor
  //Save the args array.
  this.args = args;

  if(args.length == 0){
    this.args = new String[4];
    this.args[0] = "play";//Play the melody immediately
    this.args[1] = "16";//beats per second
    this.args[2] = "GreensleevesTreble.txt";
    this.args[3] = "GreensleevesBass.txt";
  }//end if
```

Because of the code in Listing 3 , the program can be run in default mode without the requirement to enter command-line parameters. This code creates default **args** data if no parameters are entered on the command line. The code requires that the files named **GreensleevesTreble.txt** and **GreensleevesBass.txt** exist in the subfolder named **Music** .

Get a melody array

The code in Listing 4 is the same as code that you have seen in previous modules. The only thing new is that this program requires a text file containing treble clef data and optionally a file containing bass clef data in the subfolder named **Music** .

Listing 4 . Get a melody array.

Listing 4 . Get a melody array.

```
    AudioSignalGenerator02 sigGen =
                                  new
PlayerPiano01(audioParams,this.args,melody);
    melody = sigGen.getMelody();

    //Play or file the audio data
    new
AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-----------------------------------------------------------------
-----//
}//end class MusicComposer09.java
```

Listing 4 also signals the end of the constructor and the end of the class named **MusicComposer09** .

The class named PlayerPiano01

This class simulates an old-fashioned player piano and makes it possible to compose a melody by specifying the sequence of notes and the duration of each note for both the treble and bass clefs. The program uses text files to store the notes.

The notes are converted to frequencies, which are then used to produce mono or stereo sound. While not perfect, an attempt was made to cause that sound to resemble the sound of a piano.

Treble clef and bass clef

A file containing treble clef notes is required. Bass clef notes are optional. If a file is provided containing bass clef notes, the bass clef melody is emitted from the left speaker and the treble clef melody is emitted from the right speaker. If bass clef notes are not provided, the program defaults to monaural with the treble clef melody being emitted with equal volume from both speakers.

Command-line parameters

Various operating parameters are provided by way of command-line parameters. One of the command-line parameters makes it possible to play the music immediately or to write it into an audio file of type AU for playback later. The notes for the melody are specified in one or two text files. As mentioned earlier, a file containing treble clef notes is required. A file containing bass clef notes is optional.

The input file names and other parameters are specified on the command line as shown in the comments at the beginning of the source code file for the class named **MusicComposer09** .

Required text file format

Each line in the text file specifies the duration and one or more keys that would be pressed on a piano simultaneously. The items are delimited by commas, and spaces are not allowed. For example, the following lines

of text specify four instances of the A-Major chord for quarter, half, three-fourths, and whole notes with a whole note rest in between. (See http://www.true-piano-lessons.com/piano-chord-chart.html for chord chars.) Note that X is used as the symbol for silence or a musical rest.

- // This is a comment in the text file
- 1,A3,C4#,E4
- 4,X
- 2,A3,C4#,E4
- 4,X
- 3,A3,C4#,E4
- 4,X
- 4,A3,C4#,E4

Normally, the first character on each line must be a number or a /. Comments must begin in the first column with a forward slash (/). The program will ignore:

- Comment lines that begin with /
- Blank lines that result in a string with a zero length
- Lines that begin with a space

The frequency range extends from A2 (110 Hz) to A7 (3520 Hz). Note, however, that some audio speakers cannot produce sound at the low or high end of that spectrum. Middle-C *(261.63 Hz)* is specified as C4.

Shaping the notes

The amplitude versus time of each note is shaped by a scale factor. The amplitude is maximum at the beginning and decays linearly to zero at the end.

Beginning of the PlayerPiano01 class

The class named **PlayerPiano01** begins in Listing 5 . Assuming that you have studied the previous modules in this series, there is nothing in Listing 5 that should require an explanation beyond the embedded comments.

Listing 5 . Beginning of the PlayerPiano01 class.

Listing 5 . Beginning of the PlayerPiano01 class.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class PlayerPiano01 extends AudioSignalGenerator02{

  //Used to store treble clef notes as a list of arrays.
  ArrayList trebleClef = null;
  //Used to store bass clef notes as a list of arrays.
  ArrayList bassClef = null;

  //Names of input text files are stored here.
  String trebleFileName;
  String bassFileName;

  //Routine working variables
  int trebleLengthInBeats;
  int bassLengthInBeats;
```

The constructor for the PlayerPiano01 class

The constructor is shown in Listing 6 . The code in the constructor checks the length of the **args** array. Depending on the length, it gets and saves the file names for neither, either, or both of the text files that define the melody.

Listing 6 . The constructor for the PlayerPiano01 class.

Listing 6 . The constructor for the PlayerPiano01 class.

```
public PlayerPiano01(AudioFormatParameters01 audioParams,
                     String[] args,
                     byte[] melody){
  super(audioParams,args,melody);

  //Get names of files containing the durations and notes.
  if(args.length >= 3){
    //Required trebleFileName
    trebleFileName = args[2];
  }else{
    System.out.println("No trebleFileName provided");
  }//end else

  if(args.length >= 4){
    //Optional bassFileName
    bassFileName = args[3];
  }//end if
}//end constructor
```

Beginning of the getMelody method

This program uses the same structure as several programs in previous modules. By now, you should be familiar with that structure. The overridden **getMelody** method, which is inherited from the abstract class name **AudioSignalGenerator02** and called from Listing 4 , begins in Listing 7 .

This method returns a melody array that will play piano-like sounds using input musical notation like

A, A#, B, C, C#, D, D# etc.

See Listing 29 for an example of a text file containing the treble clef notes for the Greensleeves melody.

Listing 7 . Beginning of the getMelody method.

Listing 7 . Beginning of the getMelody method.

```
  byte[] getMelody(){
    if(bassFileName != null){
      //Specify stereo output.
      audioParams.channels = 2;
    }else{
      //Specify mono output.
      audioParams.channels = 1;//superfluous - same as default
    }//end else
    System.out.println("audioParams.channels = " +
audioParams.channels);

    //Controls how fast or slow the notes are played.
     int beatsPerSec = Integer.parseInt(args[1]);
```

The treble clef and bass clef data

As explained earlier, a text file containing data for the treble clef must be provided. *(Actually, it doesn't matter which clef is represented by the single required text file. It could be data for the bass clef if you want to hear that absent of the treble clef melody. It could even be random notes. However, I will continue to refer to the treble clef and the bass clef as a convenient way to distinguish between the contents of the two text files.)*

If bass data *(the optional second text file)* is provided, the output audio data will be interlaced in the output melody array so that the bass will be played through the left speaker and the treble will be played through the right speaker.

The code in Listing 7 sets the value for **channels** to either 2 for stereo or 1 for monaural.

Beats per second

Listing 7 also gets and saves the value of **beatsPerSec** from the **args** array. Recall that this value determines how fast or slow the notes will be played. A low value causes the note to be played slowly and a higher value causes the notes to be played faster.

Get and save treble clef note data

The code in Listing 8 reads the text file *(from the subfolder named **Music**)* containing the data for the treble clef and saves that data in an **ArrayList** object in a form that is suitable for processing later. *(Hopefully you are familiar with the **ArrayList** class as a result of your studies of the Collections Framework in general and the module titled Java4040: Purpose of Framework Implementations and Algorithms in particular.)*

Listing 8 . Get and save treble clef note data.

Listing 8 . Get and save treble clef note data.

```
    try{
      trebleClef = new ArrayList();
      BufferedReader in = new BufferedReader(
                                    new FileReader("Music/" +
trebleFileName));
      String str;
      while ((str = in.readLine()) != null) {
        //Split the input string into multiple substrings using the
comma as
        // the delimiter and save them in an array object.
        String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() != 0 && !
(strArr[0].substring(0,1).equals("/"))
                                    && !
(strArr[0].substring(0,1).equals(" "))){
          //Add the array to the end of the list.
          trebleClef.add(strArr);
        }//end if

      }//end while
      in.close();//close the input file
```

Consider the format of the line of text shown near the end of Listing 30 that reads:

9,A3,C4,E4

To play this note on a piano, the user would press three piano keys simultaneously and hold them down for nine beats.

Parsing the treble clef note data

The code in Listing 8 reads each line of text from the text file and parses it into two or more strings using the comma as a delimiter. The two or more strings are placed in an array object of type **String[]** and that object's reference is added to the **ArrayList** object.

Every line of text that is not ignored will contain at least two items separated by commas. The first item must be a number. This is the amount of time that the note will be held when it is played. The remaining one or more items specify the piano keys that are to be pressed simultaneously to play that note. Therefore, the arrays that are created from the lines of text in the text file are likely to be of different lengths with the minimum length of two elements. The more complicated the melody, the more likely is the occurrence of arrays with more than two elements.

The code in Listing 8 ignores:

- Blank lines that result in a string with a zero length
- Comments that begin with a forward slash character *(/)*

- Lines that begin with a space

By ignore, I mean that they are not used to create an array that is added to the end of the **ArrayList** object.

The text file with the given name in the subfolder named **Music** is opened for reading by one of the early statements in Listing 8 . It is closed by the last statement in Listing 8 .

Get and save bass clef note data

Listing 9 does essentially the same thing for the bass clef note data if a bass clef text file is specified on the command line.

Listing 9 . Get and save bass clef note data.

```
      if(bassFileName != null){
        //A file was specified for bass cleft data. Need to process it.
        bassClef = new ArrayList();
        in = new BufferedReader(new FileReader("Music/" +
bassFileName));
        while ((str = in.readLine()) != null) {
          String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() != 0 && !
(strArr[0].substring(0,1).equals("/"))
                                    && !
(strArr[0].substring(0,1).equals(" "))){
            bassClef.add(strArr);
          }//end if

        }//end while
      }//end if
    }catch(Exception ex){
      ex.printStackTrace();
    }//end catch
```

Compute the length of each clef in beats

Later on in the program, we will need to relate the notes from the text file to real time in seconds. We will also need to know the total length of the melody in seconds in order to create an array object of sufficient length to contain the audio data at a specified sampling rate.

In preparation for that requirement, Listing 10 uses an iterator to extract and add the number of beats for each note for each clef in order to get the total length of the melody in beats. *(Hopefully you are familiar with the use of an iterator as a result of your studies of the Collections Framework)*

Listing 10 . Compute the length of each clef in beats.

```
//Compute length of trebleClef in beats
//Get an iterator on the trebleClef
Iterator iter = trebleClef.iterator();
while(iter.hasNext()){
  //Extract the next array of notes from the ArrayList.
  String[] array = (String[])iter.next();
  //Get the duration in beats and add to the total.
  trebleLengthInBeats += Integer.parseInt(array[0]);
}//end while

if(bassFileName != null){
  //Compute length of bassClef in beats
  //Get an iterator on the trebleClef
  iter = bassClef.iterator();
  while(iter.hasNext()){
    //Extract the next array of notes from the ArrayList
    String[] array = (String[])iter.next();
    //Get the duration in beats and add to the total.
    bassLengthInBeats += Integer.parseInt(array[0]);
  }//end while
```

Check for treble and bass clefs of different lengths

Listing 11 continues the **if** statement that began in Listing 10 .

Given that the two text files were created separately and probably manually, there is a good possibility that the total length in beats of the treble clef data and the total length of the bass clef data are not exactly the same. Listing 11 checks for that possibility and prints out a warning if the lengths don't match. As you will see later, if they don't match, the longer of the two will be truncated to the length of the shorter one. *(This might cause you to edit the files and add some musical rests to the end of the shorter one to make them match or, find and fix a problem somewhere within the melody.)*

Listing 11 . Check for treble and bass clefs of different lengths.

Listing 11 . Check for treble and bass clefs of different lengths.

```
      if(trebleLengthInBeats != bassLengthInBeats){
        System.out.println("Treble and bass are different lengths.");
        System.out.println("Will use the shorter of the two.");
      }//end if
    }//end if
```

Convert treble notes to amplitude values

With the exception of the last statement in Listing 12 , all the code in Listing 12 is very similar to code that you have seen before.

Listing 12 . Convert treble notes to amplitude values.

Listing 12 . Convert treble notes to amplitude values.

```
    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample
rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;


    //Create an array of sufficient size to contain the treble melody.
Treat
    // as mono at this point. May combine with bass melody later to
create
    // a stereo output. Add an extra one-half second of capacity to deal
with
    // possible round off error at the end.
    byte[] trebleMelody = new byte[
                      (int)(audioParams.sampleRate/2 +
trebleLengthInBeats*

audioParams.sampleRate*bytesPerSampPerChan/beatsPerSec)];

    System.out.println("trebleMelody.length = " + trebleMelody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(trebleMelody);

    //Call a method that transforms the notes for the clef into an array
of
    // amplitude values.
    makeMusic(trebleClef,beatsPerSec);
```

The last statement in Listing 12 calls a method named **makeMusic** passing the treble note data along with the value of **beatsPerSec** as parameters. The purpose of the **makeMusic** method is to transform the note data into an array of amplitude values that can be played or written into an audio output file. This method is the most important part of the entire program. I will come back and dedicate quite a lot of time to it later.

Convert bass notes to amplitude values

Continuing with the **getMelody** method, Listing 13 does essentially the same thing with the bass clef data if a file containing bass clef data was specified as a command-line parameter.

Listing 13 . Convert bass notes to amplitude values.

Listing 13 . Convert bass notes to amplitude values.

```
    //Process the bass clef if it exists.
    if(bassFileName != null){

      //Create an array of sufficient size to contain the bass melody.
Treat
      // as mono at this point. Will combine with the trebleMelody later
to
      // create a stereo output. Add an extra one-half second of
capacity to
      // deal with possible round off error at the end.
      byte[] bassMelody = new byte[
                      (int)(audioParams.sampleRate/2 +
bassLengthInBeats*

audioParams.sampleRate*bytesPerSampPerChan/beatsPerSec)];

      System.out.println("bassMelody.length = " + bassMelody.length);

      //Prepare a ByteBuffer for use
      byteBuffer = ByteBuffer.wrap(bassMelody);

      //Call a method that transforms the notes for the clef into an
array
      // of amplitude values.
      makeMusic(bassClef,beatsPerSec);
```

Populate and return the melody array with stereo data

Listing 14 continues the **if** statement that began at the top of Listing 13 , and is executed only if the user specified a bass data file on the command line.

Listing 14 . Populate and return the melody array with stereo data.

Listing 14 . Populate and return the melody array with stereo data.

```
      //Use the shorter of the two lengths if they don't match. Note the
use
      // of a Java conditional operator to accomplish this.
      int lengthLimit = (trebleMelody.length <= bassMelody.length)
                                     ? trebleMelody.length :
bassMelody.length;

      //Create an output array that is twice the length of the shorter
of the
      // bass or treble melodies to accommodate a stereo representation
of the
      // melody. The bass melody will be put in the left-channel bytes
in the
      // array. Similarly, the treble melody will be put in the right-
channel
      // bytes in the array.
      melody = new byte[2*lengthLimit];

      //Interlace the bass and treble melody data in the array so that
the
      // bass will be played through the left speaker and the treble
will be
      // played through the right speaker.

      for(int cnt = 0;cnt < melody.length-4;cnt+=4){
        melody[cnt] = bassMelody[cnt/2];
        melody[cnt+1] = bassMelody[1 + cnt/2];
        melody[cnt+2] = trebleMelody[cnt/2];
        melody[cnt+3] = trebleMelody[1 + cnt/2];
      }//end for loop

      return melody;//return the array and terminate the method

    }//end if
```

All of the code in Listing 17 is straightforward and shouldn't require an explanation beyond the embedded comments. The **getMelody** method terminates when the **return** statement is executed near the bottom of Listing 17 . At this point, the melody array has been populated with bass audio data for the left speaker and treble audio data for the right speaker. Control returns to Listing 4 , which causes the audio data in the melody array to be played or written into an output audio file of type AU.

Populate and return the melody array with monaural data

The code in Listing 15 is executed only if the **if** statement at the beginning of Listing 13 returned false, meaning that the user did not specify a file containing bass note data on the command line. In that case, all of the code in the **if** statement is skipped, passing control to the **return** statement in Listing 15.

Listing 15 . Populate and return the melody array with monaural data.

```
    return trebleMelody;

  }//end method getMelody
```

Listing 15 returns a reference to the monaural **trebleMelody** array containing only treble clef audio data.

Listing 15 also signals the end of the **getMelody** method.

The getPiano method

At the end of Listing 12 , you saw a call to a method named **makeMusic** . I told you that the purpose of the **makeMusic** method is to transform the note data into an array of amplitude values that can be played or written into an audio output file. I also told you that the **makeMusic** method is the most important part of the entire program and that I would explain it in detail later.

The **makeMusic** method must be able to determine the frequency or pitch of every note that is specified in the text files containing the data for a melody using musical annotation. That is no small task. I wrote a utility method named **getPiano** that creates a **Hashtable** object that ties note names to note frequencies and provides that **Hashtable** object to the **makeMusic** method. That method begins in Listing 16 .

In case you have forgotten, a **Hashtable** object is a data structure that ties *keys* to *values* . *(Items in a **Hashtable** are called keys but this is unrelated to the keys on a piano keyboard.)* In this case, the *keys* in the **Hashtable** are the note names and the *values* are the frequency values that are tied to those not names. The program can ask the **Hashtable** object to provide the frequency for a given note name. For example, the piano key commonly known as middle-C is named **C4** . The program can ask the **Hashtable** for the frequency value for the key **C4** and the **Hashtable** will return 261.63 Hz.

Musical instruments must be tuned

I mentioned earlier that I am not a musician and I am clearly not a musical theoretician. What I about to report to you is what I have learned during the past few weeks while working on this program.

In order for multiple instruments to play and sound well together, they must be tuned using compatible frequencies regardless of the type of instrument. There is a set of six notes or keys, moving from left to right on the piano keyboard, where there is general agreement as to the frequency that should be tuned for each piano key:

- A2 = 110 Hz
- A3 = 220 Hz
- A4 = 440 Hz
- A5 = 880 Hz
- A6 = 1760 Hz
- A7 = 3520 Hz

Note that each frequency is double the frequency above it in this list. This doubling in frequency is known as an octave. There is one octave between A2 and A3, one octave between A3 and A4, etc. As you move to the right on the piano keyboard, the frequency doubles between a particular piano key and the next key to the right with the same name *(A, B, C, etc.)*

The in-between frequencies

While there appears to be general agreement as to the frequencies that should be assigned to the A keys, there appears to be something less than total agreement as to the frequencies for the keys in between. However, I believe that I understand the formula that is often used to tune the keys on a piano. The frequency for any key is multiplied by a factor that is the twelfth root of two *(1.05946309436)* to obtain the frequency for the key immediately to its right.

The key sequence from any A-key to the next A-key to the right is as follows:

A, A#, B, C, C#, D, D#, E, F, F#, G, G#, A

If you count the number of keys, there are twelve steps from an A-key to the next A-key to the right. That is one octave, meaning that the frequency must double from any A-key to the next A-key to the right. If you multiply the twelfth root of two by itself twelve times, the result will be two. If you multiply the frequency of any key by the twelfth root of two to get the frequency for the key to the right, and you do that twelve times, the frequency after twelve steps will have doubled.

Now that we know what we need to do, let's look at the code that will do it.

Beginning of the getPiano method

The getPiano method begins in Listing 16 . This method creates and returns a reference to a **Hashtable** object containing the name *(key)* and frequency *(value)* of every note from A2 *(110 Hz)* at the low end to A7 *(3520 Hz)* at the high end. This 61-note, five-octave range is close to the limits of the frequency range that I can hear on my computer. In fact, I can only barely hear A2. *(Click AllNotes to hear each of the 61 notes played from lowest to highest.)*

Listing 16 . Beginning of the getPiano method.

```
Hashtable getPiano(){
  Hashtable piano = new Hashtable();

  double factor = 1.05946309436;//12th root of 2

  //Define the name and frequency of the first note.
  double freq = 110;//Frequency of A2 at 110 Hz. Start with this.
  String note = "A2";//Name of note at 110 Hz. Start with this.

  //Used to parse a note into 3 single-character substrings such
  // as C, 5, and #.
  String sub1 = null;
  String sub2 = null;
  String sub3 = null;
```

The first half of the code in Listing 16 is completely straightforward.

The number in the center of the note names *(such as A3#)* changes at C and not at A. Also, the # character appears on some notes and not on all notes. As a result, it is difficult to use a loop structure to deal with the names of the notes.

I will decompose each note into two or three substrings for processing. In other words, the note name "A2" will be decomposed into an "A" and a "2". The note name "C5#" will be decomposed into a "C", a "5", and a "#". The last three variables that are declared in Listing 16 will be used to save those strings.

Store the current note and compute the frequency of the next note

The process of populating the **Hashtable** object with note names and frequencies is controlled by a **for** loop that begins in Listing 17 .

Listing 17 stores the name and frequency for the current note and computes the frequency of the next note. (*The initial name and frequency for the current note were defined in Listing 16 .)* The frequency of the next note is computed by multiplying the frequency of the current note by the twelfth root of two, which is stored in the variable named factor.

Listing 17 . Store the current note and compute the frequency of the next note.

```
    for(int cnt = 0;cnt < 61;cnt++){
      //This loop counts from A1 through A7 at 3520 Hz.
      //Store the name and the frequency of the note in the next element
in
      // the hashtable.
      piano.put(note,freq);

      //Compute the frequency of the next note
      freq *= factor;
```

Construct the name of the next note

While it may be possible to write a loop structure to deal with note names, it was not obvious to me how to do that. Therefore, I took a brute force approach and wrote a long and tedious block of logic code to do the job.

Listing 18 shows the logic used to decompose the name of the current note into two or three single-character strings and to use those strings to construct the name of the next note. That note will be the current note at the top of the next iteration and will be saved, along with the frequency in the **Hashtable** object. I will leave it as an exercise for the student to understand and possibly improve on this logic.

Listing 18 . Construct the name of the next note.

```
      //Use logic to construct the name of the next note in a sequence
such
      // as A2,A2#,B2,C3,C3#,D3,D3#,E3,F3,F3#,G3,G3#,A3, etc.
      //Begin by parsing the current note name into three single-
character
      // substrings, the third of which may be null.
      if(note.length() == 3){
        sub1 = note.substring(0,1);
        sub2 = note.substring(1,2);
        sub3 = note.substring(2,3);
      }else{
        sub1 = note.substring(0,1);
        sub2 = note.substring(1,2);
        sub3 = null;
      }//end if

      //Use the three substrings of the current note name to determine
the
      // name of the next note. This is long and tedious but it works.
      if((sub1.equals("A")) && (sub3 == null)){
        sub1 = "A";
        sub3 = "#";
      }else if((sub1.equals("A")) && (sub3.equals("#"))){
        sub1 = "B";
        sub3 = null;
      }else if((sub1.equals("B"))){
        sub1 = "C";
        sub3 = null;
        //Increment the number
        sub2 = "" + (1 + Integer.parseInt(sub2));
      }else if((sub1.equals("C")) && (sub3 == null)){
        sub1 = "C";
        sub3 = "#";
      }else if((sub1.equals("C")) && (sub3.equals("#"))){
        sub1 = "D";
        sub3 = null;
      }else if((sub1.equals("D")) && (sub3 == null)){
        sub1 = "D";
        sub3 = "#";
      }else if((sub1.equals("D")) && (sub3.equals("#"))){
        sub1 = "E";
        sub3 = null;
      }else if((sub1.equals("E"))){
        sub1 = "F";
        sub3 = null;
      }else if((sub1.equals("F")) && (sub3 == null)){
        sub1 = "F";
        sub3 = "#";
      }else if((sub1.equals("F")) && (sub3.equals("#"))){
        sub1 = "G";
        sub3 = null;
```

Listing 18 . Construct the name of the next note.

```
      }else if((sub1.equals("G")) && (sub3 == null)){
        sub1 = "G";
        sub3 = "#";
      }else if((sub1.equals("G")) && (sub3.equals("#"))){
        sub1 = "A";
        sub3 = null;
      }else{
        System.out.println("Can't reach this point.");
      }//end else


      //Construct the next note from the updated substrings.
      if(sub3 == null){
        note = sub1 + sub2;
      }else{
        note = sub1 + sub2 + sub3;
      }//end if
    }//end for loop

    return piano;
  }//end getPiano

}//end class PlayerPiano01
```

A reference to the populated **Hashtable** object referred to by piano is returned when the for loop terminates at the bottom of Listing 18 .

Listing 18 also signals the end of the **getPiano** method and the end of the class named **PlayerPiano01** . However, we aren't finished yet. We need to go back and examine the code in the method named **makeMusic** that we put on the back burner a little earlier.

Beginning of the method named makeMusic

The method named **makeMusic** begins in Listing 19 . This method transforms the notes for a clef into an array of amplitude values. This method doesn't know the difference between a treble clef and a bass clef. It is called once to transform the treble clef at the bottom of Listing 12 . It is called again to transform the bass clef in Listing 13 if the user provides a text file containing bass clef data.

Listing 19 . Beginning of the method named makeMusic.

Listing 19 . Beginning of the method named makeMusic.

```
  void makeMusic(ArrayList clef,int beatsPerSec){

    double gain = 4000.0;//Set the output volume to a reasonable level.

    //Get a hashtable that maps note names into note frequencies from A2
    // through A7. Note that the name "X" is used to indicate a period
of
    // silence or a rest, so it is appended onto the end of the
hashtable with
    // a frequency of 0.0.
    //Frequency values can be checked against
    // http://www.phy.mtu.edu/~suits/notefreqs.html.
    Hashtable piano = getPiano();
    piano.put("X",0.0);

    //Miscellaneous variables
    double freq = 0;
    int beats = 0;
    double scaleFactor = 0;
```

Nothing in Listing 19 should require an explanation beyond the embedded comments.

Process each array containing duration and note names in the ArrayList object

Listing 20 shows the beginning of a **while** loop that uses an iterator to process each array containing duration and note names in the **ArrayList** object

Listing 20 . Process each array containing duration and note names in the ArrayList object.

```
    Iterator iter = clef.iterator();

    while(iter.hasNext()){
      //Get the next array containing duration and note names
      String[] array = (String[])iter.next();
      //Get the duration of the note in beats
      beats = Integer.parseInt(array[0]);
```

The code in Listing 20 accesses the next array and then extracts the duration *(in beats)* from the array and stores it in the variable named **beats** .

Process each sample

Listing 21 shows the beginning of a **for** loop that is used to process each sample that makes up this note. The output sampling rate is constant. Therefore, notes with a short duration comprise fewer samples than notes with a long duration.

Listing 21 . Process each sample.

```
      for(int cnt = 0; cnt < beats*audioParams.sampleRate/beatsPerSec;
cnt++){
        //Compute the time for this iteration to use when evaluating the
        // cosine function.
        double time = cnt/audioParams.sampleRate;
        double sum = 0;//sum of values for this iteration
```

Process each piano key that is pressed

Recall that a human piano player can press several keys simultaneous, as in *(9,A3,C4,E4)* at the end of Listing 30 . In this case, the piano player would press three keys simultaneously on the piano keyboard and hold them down for nine beats.

Listing 22 shows a **for** loop that is used to compute the amplitude at the current time in the current sample including the possibility of two or more keys being pressed. When two or more keys are pressed simultaneously, the amplitude of the output is computed as the sum of the amplitudes of the individual keys. This code shouldn't require an explanation beyond that provided by the embedded comments. Note that the name of the key is used to extract the frequency for that key from the **Hashtable** object that was created by the method named **getPiano** that was called in Listing 19 .

Listing 22 . Process each piano key that is pressed.

Listing 22 . Process each piano key that is pressed.

```
        for(int element = 1;element < array.length;element++){
          //Iterate on the piano keys defined in the array.
          //Get the name of the next key and make sure that it is upper-
case.
          String noteName = array[element].toUpperCase();
          try{
            //Use the noteName and get the corresponding frequency from
the
            // hashtable. Note that results are retrieved from the list
as
            // type Object and must be cast to the correct type.
            freq = (double)piano.get(noteName);
          }catch(java.lang.NullPointerException ex){
            ex.printStackTrace();
            System.out.println("noteName: " + noteName);
          }//end catch

          //Compute the amplitude for this note at this time and add it
to
          // the sum unless it is a musical rest.
          if(!noteName.equals("X")){
            //This is not a musical rest.
            sum += Math.cos(2*Math.PI*(freq)*time);
          }//end if
          //Go back to the top of the loop and get the next key from the
          // array, if any.
        }//end for loop
```

Use a scale factor to shape the note

If you press a key on some electronic keyboards and hold the key down, the sound intensity will remain constant with time. However, if you press a piano key and hold it down, the sound intensity decreases over time. The code in Listing 23 attempts to achieve that same effect.

Listing 23 . Use a scale factor to shape the note.

Listing 23 . Use a scale factor to shape the note.

```
        scaleFactor = gain*((beats*audioParams.sampleRate/beatsPerSec) -
cnt)

/(beats*audioParams.sampleRate/beatsPerSec);

        //Scale the amplitude value and put it into the output array.
        byteBuffer.putShort((short)(scaleFactor*sum));
        //Go back and compute the next sample values for this set of
keys
        // until the note duration is satisfied.
      }//end for loop
      //Go back, retrieve, and process the next set of keys for a given
      // duration value.
    }//end while

  }//end makeMusic method
```

At this point in the creation of the audio sample, the amplitude values for all the keys that were pressed simultaneously have been added together. The resulting amplitude value is scaled so that each note has a maximum amplitude at the beginning and a zero amplitude when the duration has expired. A linear scale factor is used to accomplish this.

The remaining code in Listing 23 shouldn't require an explanation.

Listing 23 ends the **for** loop that began in Listing 21 , ends the **while** loop that began in Listing 20 , and signals the end of the **makeMusic** method.

Run the program

I encourage you to copy the code from Listing 24 through Listing 28 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

When you compile this program, you will probably see the following warnings:

> Note: .\PlayerPiano01.java uses unchecked or unsafe operations.

> Note: Recompile with -Xlint:unchecked for details.

You can simply ignore those warnings if you wish to do so. Otherwise, you can learn how to eliminate the warnings by doing the following:

- Go to Object-Oriented Programming (OOP) with Java
- Open Contents
- Open ITSE2317 - Java Programming (Intermediate)
- Open Generics

- Open Java4210: Getting Started with Generics
- Study that module and the seven modules following it as listed in the Contents.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2060-A Player Piano Simulator
- File: Jbs2060.htm
- Published: 08/30/14
- Revised: 09/29/15

Complete program listings

Listings of the classes discussed in this module are provided below. Listings of the musical notes for the Greensleeves melody are also provided. A listing of a Windows batch file that can be used to compile, run, and play the Greensleeves melody is also provided.

Click here to download a zip file containing the text files and the Windows batch files needed to play all of the melodies in the above list . The zip file also contains the audio files of type AU.

Listing 24 . The class named AudioFormatParameters01.

Listing 24 . The class named AudioFormatParameters01.

```
/*File AudioFormatParameters01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**********************************************************************

public class AudioFormatParameters01{
  //The following are audio format parameters used by the Java audio system.
  // They may be modified by the signal generator at runtime.  Values allowe
  // by Java SDK 1.4.1 are shown in comments.
  public float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100
  public int sampleSizeInBits = 16;
  //Allowable 8,16
  public int channels = 1;
  //Allowable 1,2
  public boolean signed = true;
  //Allowable true,false
  public boolean bigEndian = true;
  //Allowable true,false
}//end class AudioFormatParameters01
//======================================================================
```

...

Listing 25 . The class named AudioPlayOrFile01.

```
/*File AudioPlayOrFile01.java
Copyright 2014, R.G.Baldwin
Revised 08/16/14
**********************************************************************
import javax.sound.sampled.*;
import java.io.*;
import java.util.*;

public class AudioPlayOrFile01{
  //An object of this class is used to either play the sound in the array
  // named melody or to write it into an audio file of type AU.

  //The following are general instance variables used to create a
  // SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
```

Listing 25 . The class named AudioPlayOrFile01.

```
  SourceDataLine sourceDataLine;

  AudioFormatParameters01 audioParams;
  byte[] melody;
  String playOrFile;//"play" to play immediately or a fileName to write
                    // an output file of type AU.
  //-----------------------------------------------------------------------

  public AudioPlayOrFile01(AudioFormatParameters01 audioParams,
                           byte[] melody,
                           String playOrFile){//constructor

    this.audioParams = audioParams;
    this.melody = melody;
    this.playOrFile = playOrFile;
  }//end constructor
  //-----------------------------------------------------------------------

  //This method plays or files the synthetic audio data that has been genera
  // and saved in an array.
  void playOrFileData() {
    try{
      //Get an input stream on the byte array containing the data
      InputStream byteArrayInputStream = new ByteArrayInputStream(melody);

      //Get the required audio format
      audioFormat = new AudioFormat(audioParams.sampleRate,
                                    audioParams.sampleSizeInBits,
                                    audioParams.channels,
                                    audioParams.signed,
                                    audioParams.bigEndian);

      //Get an audio input stream from the ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                                     byteArrayInputStream,
                                     audioFormat,
                                     melody.length/audioFormat.getFrameSize(

      //Get info on the required data line
      DataLine.Info dataLineInfo = new DataLine.Info(SourceDataLine.class,
                                                     audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)AudioSystem.getLine(dataLineInfo);

      //Decide whether to play the audio data immediately, or to write it
      // into an audio file of type AU based on the incoming parameter named
      // playOrFile.
      if(playOrFile.toUpperCase().equals("PLAY")){
        //Create a thread to play back the data and start it running.  It wi
        // run until all the data has been played back
        new PlayAudioThread().start();
      }else{
        //Write the data to an output file with the name provided by the
```

Listing 25 . The class named AudioPlayOrFile01.

```
        // incoming parameter named playOrFile.
        try{
          AudioSystem.write(audioInputStream,
                            AudioFileFormat.Type.AU,
                            new File(playOrFile + ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//=============================================================================

  //Inner class to play back the data that was saved.
  class PlayAudioThread extends Thread{
    //This is a working buffer used to transfer the data between the
    // AudioInputStream and the SourceDataLine.  The size is rather arbitrar
    byte playBuffer[] = new byte[16384];

    public void run(){
      try{
        //Open and start the SourceDataLine
        sourceDataLine.open(audioFormat);
        sourceDataLine.start();

        int cnt;
        //Get beginning of elapsed time for playback
        long startTime = new Date().getTime();

        //Transfer the audio data to the speakers
        while((cnt = audioInputStream.read(
                                      playBuffer,0,playBuffer.length)) != -
          //Keep looping until the input read method returns -1 for empty
          // stream.
          if(cnt > 0){
            //Write data to the internal buffer of the data line where it wi
            // be delivered to the speakers in real time
            sourceDataLine.write(playBuffer, 0, cnt);
          }//end if
        }//end while

        //Block and wait for internal buffer of the SourceDataLine to become
        // empty.
        sourceDataLine.drain();


        //Get and display the elapsed time for the previous playback.
        int elapsedTime = (int)(new Date().getTime() - startTime);
        System.out.println("Elapsed time: " + elapsedTime);
```

Listing 25 . The class named AudioPlayOrFile01.

```
        //Finish with the SourceDataLine
        sourceDataLine.stop();
        sourceDataLine.close();
      }catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }//end catch

    }//end run
  }//end inner class PlayAudioThread
  //===================================================================
}//end AudioPlayOrFile01 class
```

...

Listing 26 . The class named AudioSignalGenerator02.

Listing 26 . The class named AudioSignalGenerator02.

```
/*File AudioSignalGenerator02.java
Copyright 2014, R.G.Baldwin
Revised 08/19/14

This is an abstract class that serves as the base class for several other
classes that can be used to create melodies of different types.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public abstract class AudioSignalGenerator02{

  //Note:  This class can only be used to generate signed 16-bit data.
  ByteBuffer byteBuffer;
  String[] args;
  byte[] melody;
  AudioFormatParameters01 audioParams;
  //--------------------------------------------------------------------

  //Constructor
  public AudioSignalGenerator02(AudioFormatParameters01 audioParams,
                                String[] args,
                                byte[] melody){
    this.audioParams = audioParams;
    this.args = args;
    this.melody = melody;
  }//end constructor
  //--------------------------------------------------------------------

  //The following abstract method must be overridden in a subclass for this
  // class to be of any value.
  abstract byte[] getMelody();
}//end AudioSignalGenerator02
//====================================================================
```

...

Listing 27 . The class named MusicComposer09.

Listing 27 . The class named MusicComposer09.

```
/*File MusicComposer09.java
Copyright 2014, R.G.Baldwin
Revised 08/27/14

This is a driver class for playing piano melodies. It works in
conjunction with the following classes:

AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01
PlayerPiano01

The melody to be played is defined in one and optionally two text files. The
required text file defines the notes and the duration of those notes on the
treble clef. The optional text file defines the notes and the durations of
those notes on the bass clef. Those text files must exist in a subfolder
named Music.

The melody can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with
any standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
**************************************************************************/

public class MusicComposer09{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the data for the melody that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //--------------------------------------------------------------------//

  public static void main(String[] args){
    /*Command-line parameters
    0: "play" to play immediately, fileName to create an AU file. Note that
       the output filename cannot be named play.au.
    1: Beats per second.
    2: Name of file containing treble clef data (required).
    3: Name of file containing bass clef data (optional).
    */

    //Instantiate a new object of this class.
    new MusicComposer09(args);
  }//end main
  //--------------------------------------------------------------------//

  public MusicComposer09(String[] args){//constructor
    //Save the args array.
```

Listing 27 . The class named MusicComposer09.

```
    this.args = args;

    //Create default args data if no args data is provided on the command
    // line. Requires that the files named GreensleevesTreble.txt and
    // GreensleevesBass.txt exist in a subfolder named Music.
    if(args.length == 0){
      this.args = new String[4];
      this.args[0] = "play";//Play the melody immediately
      this.args[1] = "16";//beats per second
      this.args[2] = "GreensleevesTreble.txt";
      this.args[3] = "GreensleevesBass.txt";
    }//end if

    //Requires a file containing treble clef data and optionally a file
    // containing bass clef data in a subfolder named Music.
    AudioSignalGenerator02 sigGen =
                                new PlayerPiano01(audioParams,this.args,meloc
    melody = sigGen.getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-----------------------------------------------------------------------
}//end class MusicComposer09.java
//=======================================================================
```

...

Listing 28 . The class named PlayerPiano01.

```
/*File PlayerPiano01.java
Copyright 2014, R.G.Baldwin
Revised 08/27/14

This class simulates an old-fashioned player piano and makes it possible to
compose a melody by specifying the sequence of notes and the duration of eac
note for both the treble and bass clefs.

The program uses text files to store the notes.

The notes are converted to frequencies, which are then used to produce mono
stereo sound.

A file containing treble clef notes is required. Bass clef notes are optiona
If a file is provided containing bass clef notes, the bass clef melody is
```

Listing 28 . The class named PlayerPiano01.

```
emitted from the left speaker and the treble clef melody is emitted from the
right speaker.

If bass clef notes are not provided, the program defaults to mono with the
treble clef melody being emitted with equal volume from both speakers. Vario
operating parameters are provided by way of command-line parameters. One of
the command-line parameters makes it possible to play the music immediately
to write it into an audio file of type AU for playback later.

The notes for the melody are specified in one or two text files. A file
containing treble clef notes is required. A file containing bass clef notes
optional. The input file names and other parameters are specified on the
command line as shown in the comments at the beginning of the source code fi
for the class named MusicComposer09.

Each line in the text file specifies the duration and one or more keys that
would be struck on a piano simultaneously. For example, the following lines
text specify four instances of the A-Major chord for quarter, half,
three-fourths, and whole notes with a whole note rest in between.
(See http://www.true-piano-lessons.com/piano-chord-chart.html for chords.)
Note that X is used as the symbol for silence or a musical rest.

// This is a comment in the text file
1,A3,C4#,E4
4,X
2,A3,C4#,E4
4,X
3,A3,C4#,E4
4,X
4,A3,C4#,E4

Normally, the first character must be a number or a /. Comments must begin i
the first column with a forward slash.

The program will ignore:

 Blank lines that result in a string with a zero length
 Lines that begin with a space

The range extends from A2 (110 Hz) to A7 (3520 Hz). Note, however, that many
audio speakers cannot produce sound at the low or high end of that spectrum.

Middle-C (261.63 Hz) is specified as C4.

The amplitude versus time of each note is shaped by a scale factor. The
amplitude is maximum at the beginning and decays linearly to zero at the end

You should be able to play the audio file back with any standard media playe
that can handle the AU file type.
*******************************************************************/
import java.io.*;
import java.nio.*;
import java.util.*;
```

Listing 28 . The class named PlayerPiano01.

```
public class PlayerPiano01 extends AudioSignalGenerator02{

  //Used to store treble clef notes as a list of arrays.
  ArrayList trebleClef = null;
  //Used to store bass clef notes as a list of arrays.
  ArrayList bassClef = null;

  //Names of input text files are stored here.
  String trebleFileName;
  String bassFileName;

  //Routine working variables
  int trebleLengthInBeats;
  int bassLengthInBeats;

  //Constructor
  public PlayerPiano01(AudioFormatParameters01 audioParams,
                       String[] args,
                       byte[] melody){
    super(audioParams,args,melody);

    //Get names of files containing the durations and notes.
    if(args.length >= 3){
      //Required trebleFileName
      trebleFileName = args[2];
    }else{
      System.out.println("No trebleFileName provided");
    }//end else

    if(args.length >= 4){
      //Optional bassFileName
      bassFileName = args[3];
    }//end if
  }//end constructor
  //---------------------------------------------------------------------


  //This method returns a melody array that will play piano-like sounds usir
  // input nomenclature like A, A#, B, C, C#, D, D# etc.
  byte[] getMelody(){

    //Treble data must be provided. If bass data is also provided, the outpu
    // audio data will be interlaced in the output melody array so that the
    // bass will be played through the left speaker and the treble will be
    // played through the right speaker.
    if(bassFileName != null){
      //Specify stereo output.
      audioParams.channels = 2;
    }else{
      //Specify mono output.
      audioParams.channels = 1;//superfluous - same as default
    }//end else
    System.out.println("audioParams.channels = " + audioParams.channels);
```

Listing 28 . The class named PlayerPiano01.

```
    //Controls how fast or slow the notes are played.
    int beatsPerSec = Integer.parseInt(args[1]);


    //Read the input files to define the  that is to be played or
    // filed. They must be stored in the subfolder named Music.
    try{
      trebleClef = new ArrayList();
      BufferedReader in = new BufferedReader(
                                  new FileReader("Music/" + trebleFileName
      String str;
      while ((str = in.readLine()) != null) {
        //Split the input string into multiple substrings using the comma as
        // the delimiter and save them in an array object.
        String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() != 0 && !(strArr[0].substring(0,1).equals("/")
                                  && !(strArr[0].substring(0,1).equals(" ")
          //Add the array to the end of the list.
          trebleClef.add(strArr);
        }//end if

      }//end while
      in.close();//close the input file

      if(bassFileName != null){
        //A file was specified for bass cleft data. Need to process it.
        bassClef = new ArrayList();
        in = new BufferedReader(new FileReader("Music/" + bassFileName));
        while ((str = in.readLine()) != null) {
          String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() != 0 && !(strArr[0].substring(0,1).equals("/")
                                  && !(strArr[0].substring(0,1).equals(" ")
            bassClef.add(strArr);
          }//end if

        }//end while
      }//end if
    }catch(Exception ex){
      ex.printStackTrace();
    }//end catch

    //Compute length of trebleClef in beats
    //Get an iterator on the trebleClef
    Iterator iter = trebleClef.iterator();
```

Listing 28 . The class named PlayerPiano01.

```
    while(iter.hasNext()){
      //Extract the next array of notes from the ArrayList.
      String[] array = (String[])iter.next();
      //Get the duration in beats and add to the total.
      trebleLengthInBeats += Integer.parseInt(array[0]);
    }//end while

    if(bassFileName != null){
      //Compute length of bassClef in beats
      //Get an iterator on the trebleClef
      iter = bassClef.iterator();
      while(iter.hasNext()){
        //Extract the next array of notes from the ArrayList
        String[] array = (String[])iter.next();
        //Get the duration in beats and add to the total.
        bassLengthInBeats += Integer.parseInt(array[0]);
      }//end while

      //Check for the possibility that the treble clef and the bass clef
      // are different lengths. If so, will play using the shorter of the
      // two later. This could a portion of the end of the melody to not
      // be played.
      if(trebleLengthInBeats != bassLengthInBeats){
        System.out.println("Treble and bass are different lengths.");
        System.out.println("Will use the shorter of the two.");
      }//end if
    }//end if

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;


    //Create an array of sufficient size to contain the treble melody. Treat
    // as mono at this point. May combine with bass melody later to create
    // a stereo output. Add an extra one-half second of capacity to deal wit
    // possible round off error at the end.
    byte[] trebleMelody = new byte[
                    (int)(audioParams.sampleRate/2 + trebleLengthInBeats*
                     audioParams.sampleRate*bytesPerSampPerChan/beatsPerSec

    System.out.println("trebleMelody.length = " + trebleMelody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(trebleMelody);

    //Call a method that transforms the notes for the clef into an array of
    // amplitude values.
    makeMusic(trebleClef,beatsPerSec);

    //Process the bass clef if it exists.
```

Listing 28 . The class named PlayerPiano01.

```
    if(bassFileName != null){

      //Create an array of sufficient size to contain the bass melody. Treat
      // as mono at this point. Will combine with the trebleMelody later to
      // create a stereo output. Add an extra one-half second of capacity to
      // deal with possible round off error at the end.
      byte[] bassMelody = new byte[
                     (int)(audioParams.sampleRate/2 + bassLengthInBeats*
                     audioParams.sampleRate*bytesPerSampPerChan/beatsPerSec

      System.out.println("bassMelody.length = " + bassMelody.length);

      //Prepare a ByteBuffer for use
      byteBuffer = ByteBuffer.wrap(bassMelody);

      //Call a method that transforms the notes for the clef into an array
      // of amplitude values.
      makeMusic(bassClef,beatsPerSec);

      //Use the shorter of the two lengths if they don't match. Note the use
      // of a Java conditional operator to accomplish this.
      int lengthLimit = (trebleMelody.length <= bassMelody.length)
                                    ? trebleMelody.length : bassMelody.leng

      //Create an output array that is twice the length of the shorter of th
      // bass or treble melodies to accommodate a stereo representation of t
      // melody. The bass melody will be put in the left-channel bytes in th
      // array. Similarly, the treble melody will be put in the right-channe
      // bytes in the array.
      melody = new byte[2*lengthLimit];

      //Interlace the bass and treble melody data in the array so that the
      // bass will be played through the left speaker and the treble will be
      // played through the right speaker.

      for(int cnt = 0;cnt < melody.length-4;cnt+=4){
        melody[cnt] = bassMelody[cnt/2];
        melody[cnt+1] = bassMelody[1 + cnt/2];
        melody[cnt+2] = trebleMelody[cnt/2];
        melody[cnt+3] = trebleMelody[1 + cnt/2];
      }//end for loop

      return melody;//return the array and terminate the method

    }//end if

    //There was no bass clef data. Return the trebleMelody to be played monc
    // from both speakers equally.
    return trebleMelody;

  }//end method getMelody
  //----------------------------------------------------------------------

  //This method transforms the notes for a clef into an array of amplitude
```

Listing 28 . The class named PlayerPiano01.

```
  // values.
  void makeMusic(ArrayList clef,int beatsPerSec){

    double gain = 4000.0;//Set the output volume to a reasonable level.

    //Get a hashtable that maps note names into note frequencies from A2
    // through A7. Note that the name "X" is used to indicate a period of
    // silence or a rest, so it is appended onto the end of the hashtable wi
    // a frequency of 0.0.
    //Frequency values can be checked against
    // http://www.phy.mtu.edu/~suits/notefreqs.html.
    Hashtable piano = getPiano();
    piano.put("X",0.0);

    //Miscellaneous variables
    double freq = 0;
    int beats = 0;
    double scaleFactor = 0;

    //Get an iterator on the clef
    Iterator iter = clef.iterator();

    //Process each array containing duration and note names in the list.
    while(iter.hasNext()){
      //Get the next array containing duration and note names
      String[] array = (String[])iter.next();
      //Get the duration of the note in beats
      beats = Integer.parseInt(array[0]);

      //Transform the notes into sound data at the appropriate frequencies a
      // duration.
      for(int cnt = 0; cnt < beats*audioParams.sampleRate/beatsPerSec; cnt++
        //Compute the time for this iteration to use when evaluating the
        // cosine function.
        double time = cnt/audioParams.sampleRate;
        double sum = 0;//sum of values for this iteration

        //Process each note in the array.
        for(int element = 1;element < array.length;element++){
          //Iterate on the notes defined in the array.
          //Get the name of the next note and make sure that it is upper-cas
          String noteName = array[element].toUpperCase();
          try{
            //Use the noteName and get the corresponding frequency from the
            // hashtable. Note that results are retrieved from the list as
            // type Object and must be cast to the correct type.
            freq = (double)piano.get(noteName);
          }catch(java.lang.NullPointerException ex){
            ex.printStackTrace();
            System.out.println("noteName: " + noteName);
          }//end catch

          //Compute the amplitude for this note at this time and add it to
          // the sum unless it is a musical rest.
```

Listing 28 . The class named PlayerPiano01.

```
          if(!noteName.equals("X")){
            //This is not a musical rest.
            sum += Math.cos(2*Math.PI*(freq)*time);
          }//end if
          //Go back to the top of the loop and get the next note from the
          // array, if any.
        }//end for loop

        //Amplitude values for all the notes at this time have been added ir
        // the sum.
        //Scale the amplitude value so that each note has a maximum amplituc
        // at the beginning and a zero amplitude at the end of the note. Use
        // a linear scale factor.
        scaleFactor = gain*((beats*audioParams.sampleRate/beatsPerSec) - cnt
                                /(beats*audioParams.sampleRate/beatsPerSec)

        //Scale the amplitude value and put it into the output array.
        byteBuffer.putShort((short)(scaleFactor*sum));
        //Go back and compute the next sample values for this set of notes
        // until the note duration is satisfied.
      }//end for loop
      //Go back, retrieve, and process the next set of notes for a given
      // duration value.
    }//end while

  }//end makeMusic method
  //----------------------------------------//

  //This method creates and returns a hashtable containing the name and
  // frequency of every note from A2 (110 Hz) at the low end to A7 (3520 Hz)
  // at the high end. A7 is the highest A-note that I can hear on my compute

  Hashtable getPiano(){
    Hashtable piano = new Hashtable();

    double factor = 1.05946309436;//12th root of 2
    double freq = 110;//Frequency of A2 at 110 Hz. Start with this.
    String note = "A2";//Name of note at 110 Hz. Start with this.

    //Used to parse a note into 3 single-character substrings such
    // as C, 5, and #.
    String sub1 = null;
    String sub2 = null;
    String sub3 = null;

    for(int cnt = 0;cnt < 61;cnt++){
      //This loop counts up through A7 at 3520 Hz.
      //Store the name and the frequency of the note in the next element in
      // the hashtable.
      piano.put(note,freq);

      //Compute the frequency of the next note
      freq *= factor;
```

Listing 28 . The class named PlayerPiano01.

```
      //Use logic to determine the name of the next note in a sequence such
      // as A2,A2#,B2,C3,C3#,D3,D3#,E3,F3,F3#,G3,G3#,A3
      //Begin by parsing the current note name into three single-character
      // substrings, the third of which may be null.
      if(note.length() == 3){
        sub1 = note.substring(0,1);
        sub2 = note.substring(1,2);
        sub3 = note.substring(2,3);
      }else{
        sub1 = note.substring(0,1);
        sub2 = note.substring(1,2);
        sub3 = null;
      }//end if

      //Use the three substrings of the current note name to determine the
      // name of the next note. This is long and tedious but it works.
      if((sub1.equals("A")) && (sub3 == null)){
        sub1 = "A";
        sub3 = "#";
      }else if((sub1.equals("A")) && (sub3.equals("#"))){
        sub1 = "B";
        sub3 = null;
      }else if((sub1.equals("B"))){
        sub1 = "C";
        sub3 = null;
        //Increment the number
        sub2 = "" + (1 + Integer.parseInt(sub2));
      }else if((sub1.equals("C")) && (sub3 == null)){
        sub1 = "C";
        sub3 = "#";
      }else if((sub1.equals("C")) && (sub3.equals("#"))){
        sub1 = "D";
        sub3 = null;
      }else if((sub1.equals("D")) && (sub3 == null)){
        sub1 = "D";
        sub3 = "#";
      }else if((sub1.equals("D")) && (sub3.equals("#"))){
        sub1 = "E";
        sub3 = null;
      }else if((sub1.equals("E"))){
        sub1 = "F";
        sub3 = null;
      }else if((sub1.equals("F")) && (sub3 == null)){
        sub1 = "F";
        sub3 = "#";
      }else if((sub1.equals("F")) && (sub3.equals("#"))){
        sub1 = "G";
        sub3 = null;
      }else if((sub1.equals("G")) && (sub3 == null)){
        sub1 = "G";
        sub3 = "#";
      }else if((sub1.equals("G")) && (sub3.equals("#"))){
        sub1 = "A";
        sub3 = null;
```

Listing 28 . The class named PlayerPiano01.

```
      }else{
        System.out.println("Can't reach this point.");
      }//end else


      //Construct the next note from the updated substrings.
      if(sub3 == null){
        note = sub1 + sub2;
      }else{
        note = sub1 + sub2 + sub3;
      }//end if
    }//end for loop

    return piano;
  }//end getPiano

}//end class PlayerPiano01
//=============================================================================
```

...

Listing 29 . The file named GreensleevesTreble.txt

```
//GreensleevesTreble
//0
3,A4
//1
6,C5
3,D5
//2
5,E5
1,F5
3,E5
//3
6,D5
3,B4
//4
5,G4
1,A4
3,B4
//5
6,C5
3,A4
//6
5,A4
1,G4#
```

Listing 29 . The file named GreensleevesTreble.txt

```
3,A4
//7
6,B4
3,G4#
//8
6,E4
3,A4
//9
6,C5
3,D5
//10
5,E5
1,F5
3,E5
//11
6,D5
3,B4
//12
5,G4
1,A4
3,B4
//13
5,C5
1,B4
3,A4
//14
5,G4#
1,F4#
3,G4#
//15
9,A4
//16
9,A4
//17
```

...

Listing 30 . The file named GreensleevesBass.txt

```
//GreensleevesBass
//0
3,x
//1
6,A3
3,A3
//2
```

Listing 30 . The file named GreensleevesBass.txt

```
6,A3
3,E4
//3
6,G3
3,D4
//4
6,G3
3,D4
//5
6,F3
3,C4
//6
6,F3
3,C4
//7
6,E3
3,B3
//8
6,E3
3,B3
//9
6,A3
3,E4
//10
6,G3
3,D4
//11
6,G3
3,D4
//12
6,G3
3,D4
//13
6,F3
3,C4
//14
6,E3
3,B3
//15
3,A3
3,C4
3,E4
//16
9,A3,C4,E4
//17
```

...

Listing 31 . The file named Greensleeves.bat.

```
echo off
del *.class

del Greensleeves.au

echo on
javac MusicComposer09.java
java MusicComposer09 play 8 GreensleevesTreble.txt GreensleevesBass.txt
java MusicComposer09 Greensleeves 8 GreensleevesTreble.txt
GreensleevesBass.txt

echo off
del *.class

pause
```

-end-

Jbs2070-A General Purpose AudioGraph Program
This module develops a general purpose AudioGraph program that reads an input text file containing numeric values for y as a function of x and produces an output melody that represents a graph of that data. The material is presented in a format that is accessible to blind students.

Table of Contents

Preface

This module is part of a collection titled **Accessible Objected-Oriented Programming Concepts for Blind Students using Java** . It develops a general purpose *AudioGraph* program that reads an input text file containing numeric values for y as a function of x and produces an output melody that represents a graph of that data. The material is presented in a format that is accessible to blind students.

Some sample AudioGraph melodies produced by the program are listed below. The function that was evaluated to produce the audio file is shown with the (^) character indicating exponentiation.

(You should be able to play these audio files with any standard media player that can handle the AU file type. In case you are on the OpenStax site and you are unable to download the audio files, click the ***Legacy Site*** *link at the top of this page to switch over to the same module on the Legacy site. You should be able to download the* ***audio files from ther*** *e.)*

- Cubic01 : y = x^3 - 50
- Cubic02 : y = x^3 + 5*x^2 - 29*x -105 with three real roots
- DampedSinusoid01 : y = e^(-x/3.5)*cos(2*pi*f*x)
- ExponentialDamper01 : y = e^(-x/3.5)
- Parabola01 : y = x^2 - 50
- Quadratic01 : y = x^2 - 2*x - 15 with two real roots
- Sinc01 : y = sin(2*pi*f*x)/x
- Sinusoid01 : y = sin(2*pi*f*x)
- StraightLine01 : y = 2*x bipolar result
- StraightLine02 : y = 2*x + 500 all positive results

I will comment further on some of these audio files later in this module.

Click here to download a zip file containing the source code, the text files, and the Windows batch files needed to create, compile, and play your own version of these AudioGraph melodies. The zip file also contains the audio files of type AU listed above.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find the listings while you are reading about them.

Listings

- Listing 10 . Read the input text file.
- Listing 11 . The remainder of the constructor for the class named AudioGraph01.
- Listing 12 . Beginning of the method named getMelody.
- Listing 13 . Determine the frequency and data ranges.
- Listing 14 . Determine the frequency to represent a y-value of zero.
- Listing 15 . Compute audio sample values and deposit them in the output array.
- Listing 16 . The program named Sinc01.
- Listing 17 . The file named Sinc01Solver.bat.
- Listing 18 . The file named Sinc01Player.bat
- Listing 19 . The class named MusicComposer10.
- Listing 20 . The class named AudioGraph01.

Figures

Please see the section titled Not a substitute for an embossed image for a discussion of the images that are provided in this module.

- Figure 1 . Cubic01.
- Figure 2 . Cubic02.
- Figure 3 . DampedSinusoid01.
- Figure 4 . ExponentialDamper01.
- Figure 5 . Parabola01.
- Figure 6 . Quadratic01.
- Figure 7 . Sinc01.
- Figure 8 . Sinusoid01.
- Figure 9 . StraightLine01
- Figure 10 . StraightLine02.

General background information

What is an AudioGraph ?

A previous module titled Jbs2040-An Audio Graph of a Sinusoid introduced you to the concept of an *AudioGraph* . In a nutshell, an AudioGraph is a melody consisting of audio pulses that represent points on the graph of a function. Those points represent values of y as a function of x between two limits on x.

Two different AudioGraph programs were presented in earlier modules. Each was specifically designed for a particular function. One program produced an AudioGraph of a square wave. The other produced an AudioGraph of a sinusoid.

A general purpose AudioGraph program

This module expands on that concept and develops a general purpose program named **MusicComposer10** that can produce an AudioGraph for virtually any single-valued function of the independent variable x for which the values of y can be encapsulated in a simple text file. Several examples are provided above . *(By single-valued function in this context, I mean a function where, for every value of x within the range of interest, one and only one finite value exists for y.)*

Where do the text files come from ?

The program doesn't know and doesn't care where the values in the text file originate. For example, they could represent the temperature in your dorm room taken manually every hour for 36 hours and entered into a text file

using a simple text editor program. On the other hand, they could be produced by a computer program that is designed to evaluate a specific function and to write the results into an output text file as was the case for Sinc01 .

A template for evaluating functions

In addition to the AudioGraph generator program named **MusicComposer10** , this module also develops a simple template for writing programs to evaluate functions and to write the results into a text file. The zip file mentioned earlier contains the source code for programs to evaluate all of the functions in the above list . This template is so simple that a student with minimal programming knowledge could easily write her own programs to evaluate other functions. I will explain the template later .

The audio pulse rate

You may have noticed that the output pulses in most of the audio files in the above list are delivered to the speakers at a fairly rapid rate. The output pulse rate *(in pulses per second)* is specified by the user as a command-line parameter when the program is run. *(Most of the examples in the above list were run at 12 pulses per second but Sinusoid01 was run at only 5 pulses per second.)* An overview of the output from a function can be achieved using a fast output pulse rate. A more detailed audio analysis of the output can then be achieved by running the program with a slower output rate. And, if you need to do detailed numeric analysis, the numeric data is available for examination in the corresponding text file.

Play or file ?

Another user input parameter specifies whether the melody is to be played immediately or is to be saved in an audio file for playback later. Saving it in an audio file makes it very convenient to listen to over and over to allow the information content to "sink into" the brain. In addition, a student can build up a library of audio files for different functions for review later, such as when studying for a test.

Not a substitute for an embossed image

An AudioGraph is not a substitute for an embossed image of a graph. However, it may be much quicker and easier to produce, it may tell you if you need to take the time and effort to create an embossed image, and it may enhance the learning process even if you do produce an embossed image.

For the benefit of those students who have the ability to produce embossed images, the Figures section provides an image for each AudioGraph listed above . In addition, the downloadable zip file contains the JPEG files for those images.

Discussion and sample code

The program named Sinc01

Since this module is all about evaluating functions, I will begin with an explanation of the template that I have developed to evaluate a function and to write the results of the evaluation into an output text file. A complete listing of the program named Sinc01 is provided in Listing 16 . As is my custom, I will break the program down and explain it in fragments.

This program produces an output text file containing values for y as a function of x for the so called **sinc** function *(more on this later)* .

The contents of the text file can be converted into audio and either played or written into an audio file of type AU for playback later using the program named **AudioGraph01** .

A sinc function

The function that is evaluated by this program is the classic sin(x)/x function, otherwise known a the **sinc** function. It appears frequently in digital signal processing (DSP) and is one of my favorites.

To use this program as a template and modify it to handle other functions, you only need to modify the instance variables in the class named **Runner** and modify the code in the method named **getYval** . You should not modify any of the other code.

Beginning of the program named Sinc01

The beginning of the program named **Sinc01** is shown in Listing 1 . This is the driver class containing the **main** method. There is nothing new or unusual about this code. If you use this program as a template for a new program to evaluate a different function, you should not modify the code in Listing 1 .

Listing 1 . Beginning of the program named Sinc01.

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class Sinc01{//Driver class
  public static void main(String[] args){
    //Do not modify the code in this method.
    Runner obj = new Runner();
    obj.run();
  }//end main
}//end class Sinc01
```

Beginning of the class named Runner

The code in Listing 1 instantiates a new object of the class named **Runner** . The class named **Runner** begins in Listing 2 .

The first three instance variables in Listing 2 declare and initialize three variables that control the range of x for which the function will be evaluated, and the incremental steps in x for which the function will be evaluated. You may or may not want to change these values when writing a program to evaluate a different function. For example you might want to evaluate the function only for positive values of x, in which case you would probably set the value for **xMin** to 0.

The last statement in Listing 2 establishes the name of the text file that will be written into the subfolder named **Data** to contain the results of the evaluation. You probably will want to modify this to cause the name of the output file to be descriptive of the function being evaluated.

Listing 2 . Beginning of the class named Runner.

```
class Runner{
  //Modify the following instance variables as needed.
  double xMin = -20;//Minimum value for x
  double xMax = 20;//Maximum value for x
  double xInc = 0.25;//Used to determine x-values for evaluation of y-
value

  String fileName = "Data/Sinc01.txt";//Output file name in Data folder
```

The method named run

After instantiating a new object of the **Runner** class, the code in Listing 1 calls the **run** method on that new object. The **run** method is shown in its entirety in Listing 3 . You should not modify the code in the **run** method

The code in Listing 3 is straightforward. It repeatedly calls a method named **getYval** , once for each incremental value of x between the limits specified in Listing 2 . It creates an output string by concatenating the values returned by **getYval** , separating those values by commas with no spaces. When **getYval** has been called once for each incremental value of x within the specified limits, the method named **writeOutputFile** is called to write the string into a text file with the name specified in Listing 2 . You can view the method named **writeOutputFile** in Listing 16 . You should not modify the method named **writeOutputFile** .

Listing 3 . The method named run.

Listing 3 . The method named run.

```
  void run(){
    //Do not modify the code in this method.
    double xVal = xMin;
    String outString = "";
    while(xVal <= xMax){
      //Construct an output string contain comma-separated values of y
as a
      // function of x.
      outString += getYval(xVal) + ",";
      //Increment x
      xVal += xInc;
    }//end while loop

    writeOutputFile(fileName,outString);

  }//end run method
```

Beginning of the method named getYval

Listing 4 shows the beginning of the method named **getYval** .

This is the method that evaluates the function. This is where you will make the most important changes to the code if you use this program as a template for other functions. You will need to modify the code in Listing 4 to cause it to properly evaluate your new function.

As I mentioned earlier, the function evaluated in this program is the classic sin(x)/x function, otherwise known a the **sinc** function. I chose it for this explanation because it is the most complicated of the functions illustrated by the audio files in the above list .

To understand the code in Listing 4 , you need to know how to use the **sin** method of the **Math** class.

The most complicated thing about this code is illustrated by the code in the **if-else** structure. As you can see, the code in the **if** clause divides something by **xVal** , which is the incoming value of x. This value can be zero, which will result in a *"divide by zero"* error. To eliminate that possibility, the **else** clause substitutes a very small but non-zero value for **xVal** when **xVal** is actually zero. This doesn't change the audio output, but it does prevent the division by zero problem.

Listing 4 . Beginning of the method named getYval.

Listing 4 . Beginning of the method named getYval.

```
double getYval(double xVal){
  //Evaluate the function here
  double f = 0.25;
  double result = 0;
  if(xVal != 0.0){
    result = Math.sin(2*Math.PI*f*xVal)/xVal;
  }else{
    //Don't divide by zero
    result = Math.sin(2*Math.PI*f*0.00001)/0.00001;
  }//end else
```

To reiterate, you will modify the code in that portion of the **getYval** method shown in Listing 4 if you use this program as a template for a new program for a different function.

The end of the getYval method

The remainder of the **getYval** method is shown in Listing 5 . You should not modify this code.

All of the computations are being performed as type **double** . A **double** value can produce many more digits to the right of the decimal point than are useful when converting the output values to audio. The code in Listing 5 limits the number of digits to the right of the decimal point to no more than three digits. I will leave it as an exercise for the student to look up the behavior of the **rint** method of the **Math** class and ponder on how the code in Listing 5 limits the number of decimal digits in the **return** value to no more than three digits.

Listing 5 . The end of the getYval method.

```
  //Do not modify this code.
  //Limit the return value to three decimal digits
  return (Math.rint(1000.0*result))/1000.0;
}//end getY
```

Using an AudioGraph

Although it is not possible to make quantitative measurements of the **sinc** function by listening to Sinc01 , it is **possible to deduce important facts** about the function, even at a high output pulse rate. *(If you need to make quantitative measurements, the data is available for that purpose in the output text file.)* You can reach the following conclusions by listening to Sinc01 *(or a slowed-down version)* .

1. The y value oscillates above and below the horizontal or zero axis much like an ordinary sinusoid.

2. The magnitude of the oscillations builds up with time reaching a maximum near the center of the graph when the sound emitted by each speaker is about equal. After that, the magnitude of the oscillations decreases with time.
3. You can hear the zero crossings because a special sound is emitted whenever the function evaluates to zero.
4. If you were to slow the output down to about three pulses per second, you could count the pulses and determine the exact values for x at which the zero crossings occur. *(You could probably use this approach to find the roots of Cubic02 and Quadratic01 .)*
5. You would learn that the zero crossings occur every eight pulses most of the time for this sinc function.
6. You would learn that there is no zero crossing at an x value of zero. Instead, the maximum value for y occurs for an x value of zero. There are sixteen pulses between zero crossings at the center of the graph.
7. Insofar as zero crossing is concerned, you would learn that the function is symmetric about an x value of zero.
8. If you have a good ear for memorizing a melody, you would learn that the function is symmetric about zero. The values for y on the positive side of zero are a mirror image of the values for y on the negative side of zero.

Contents of the output file named Sinc01.txt

Listing 6 shows the output data values for this sinc function. *(Note that the actual output from the program is a single long string. I manually inserted line breaks every eight values to force the material to fit in this narrow presentation format. That also matches up with the zero crossings mentioned above.)*

If you examine this data, you will see that it supports the conclusions that were reached above based solely on the audio. For example, the first value on every line except the eleventh line is either 0.0 or -0.0. On that line, the first value is 1.571, which is the largest value in the entire set of values. That value occurs at the center of the set of values and matches the peak frequency in the audio.

The values on both sides of that value are 1.531. This suggests that the symmetry conclusion reached above is probably correct. Further comparison of the corresponding values will confirm the symmetry and mirror image conclusion reached above .

Listing 6 . Contents of the output file named Sinc01.txt.

Listing 6 . Contents of the output file named Sinc01.txt.

```
-0.0,-0.019,-0.036,-0.048,-0.053,-0.049,-0.038,-0.021,
0.0,0.022,0.04,0.054,0.059,0.055,0.043,0.024,
-0.0,-0.024,-0.046,-0.061,-0.067,-0.063,-0.049,-0.027,
0.0,0.028,0.052,0.07,0.077,0.072,0.057,0.031,
-0.0,-0.033,-0.061,-0.082,-0.091,-0.086,-0.067,-0.037,
0.0,0.039,0.074,0.1,0.111,0.106,0.083,0.046,
-0.0,-0.049,-0.094,-0.127,-0.143,-0.137,-0.109,-0.061,
0.0,0.067,0.129,0.176,0.2,0.195,0.157,0.09,
-0.0,-0.102,-0.202,-0.284,-0.333,-0.336,-0.283,-0.17,
0.0,0.219,0.471,0.739,1.0,1.232,1.414,1.531,
1.571,1.531,1.414,1.232,1.0,0.739,0.471,0.219,
0.0,-0.17,-0.283,-0.336,-0.333,-0.284,-0.202,-0.102,
-0.0,0.09,0.157,0.195,0.2,0.176,0.129,0.067,
0.0,-0.061,-0.109,-0.137,-0.143,-0.127,-0.094,-0.049,
-0.0,0.046,0.083,0.106,0.111,0.1,0.074,0.039,
0.0,-0.037,-0.067,-0.086,-0.091,-0.082,-0.061,-0.033,
-0.0,0.031,0.057,0.072,0.077,0.07,0.052,0.028,
0.0,-0.027,-0.049,-0.063,-0.067,-0.061,-0.046,-0.024,
-0.0,0.024,0.043,0.055,0.059,0.054,0.04,0.022,
0.0,-0.021,-0.038,-0.049,-0.053,-0.048,-0.036,-0.019,
-0.0,
```

Thus, the audio output provides an overview of the general shape of the graph. The data output is available for detailed quantitative analysis.

The file named Sinc01Solver .bat

Continuing with the explanation of Sinc01 , Listing 17 shows the contents of the file named **Sinc01Solver.bat** . *(This is a Windows batch file. A file with a similar purpose on other operating systems would be different.)* If you have the Java Development Kit installed on your *(Windows)* computer, execution of this batch file will compile and execute the file named **Sinc01.java** . This will create the text file containing the data that is needed by the AudioGraph program named **MusicComposer10** .

The file named Sinc01Player .bat

Listing 18 shows the contents of the file named **Sinc01Player.bat** . *(Again, this is a Windows batch file.)* If you have the Java Development Kit installed on your *(Windows)* computer, execution of this batch file will compile the file named **MusicComposer10** and execute it twice. The first execution will cause the audio to be played immediately. The second execution will cause the audio to be saved in a file named **Sinc01.au** .

The zip file that you can download contains all of the files mentioned above for all of the functions illustrated by the audio in the above list . Simply extract the contents of the zip file into an empty folder, execute the **...Solver.bat** files to create the data for the functions. Execute the **...Player.bat** files to hear the audio for the functions and to write the audio into output audio files of type AU.

The class named MusicComposer10

This program requires the following five classes in the same folder. Source code files for all five classes are provided in the zip file that you can download.

- AudioSignalGenerator02
- AudioPlayOrFile01
- AudioFormatParameters01
- MusicComposer10
- AudioGraph01

The first three classes in the above list have been used in many previous modules. Therefore, I won't discuss them further in this module. However, I will explain the last two classes in the list.

A complete listing of the class named **MusicComposer10** is provided in Listing 19 . As usual, I will break the class down and explain it in fragments.

This is a general purpose AudioGraph program that reads an input text file containing numeric values for y as a function of equally spaced values for x and produces an output melody that represents a graph of that data. The values for y are read as a comma-delimited list of values and are treated as type **double** . The name of the text file is input as a command-line parameter. Additional details will be provided later in the explanation of the class named **AudioGraph01** .

The melody can be played immediately or can be saved in an audio file of type AU for playback later. You should be able to play the audio file with any standard media player that can handle the AU file type

Beginning of the class named MusicComposer10

The class named **MusicComposer10** begins in Listing 7 . Note the description of the command-line parameters in Listing 7 . Otherwise, there is nothing new or unusual about the code in Listing 7 so there should be no need for further explanation beyond the embedded comments.

Listing 7 . Beginning of the class named MusicComposer10.

Listing 7 . Beginning of the class named MusicComposer10.

```
public class MusicComposer10{
  //Instantiate an object containing audio format parameters with
predefined
  // values. They may be modified by the signal generator at runtime.
Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class
definition.
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //-----------------------------------------------------------------
-----//

  //Command-line parameter (three parameters required)
  //0 - If "play", the sound will be played immediately. Otherwise, the
string
  //    will be used as a filename for an audio file of type AU. In the
latter
  //    case, it must be a string that would be valid as a file name for
the
  //    operating system in use.
  //1 - Output pulses per second
  //2 - Input file name

  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer10(args);
  }//end main
```

The constructor for the class named MusicComposer10

The constructor for the class named **MusicComposer10** is provided in Listing 8 . There is nothing new or unusual about this code so it shouldn't require further explanation.

Listing 8 . The constructor for the class named MusicComposer10.

Listing 8 . The constructor for the class named MusicComposer10.

```
  public MusicComposer10(String[] args){//constructor
    //Save the args array.
    this.args = args;

    //Create default args data if no args data is provided on the
command line.
    if(args.length == 0){
      this.args = new String[3];
      this.args[0] = "play";//Play the melody immediately
      this.args[1] = "6";//Pulses per minute
      this.args[2] = "TestData01.txt";
    }//end if

    //Get a populated array containing audio data.
    AudioGraph01 audioGraph01 = new
AudioGraph01(audioParams,this.args,melody);
    melody = audioGraph01.getMelody();

    //Play or file the audio data
    new
AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-------------------------------------------------------------------
-----//
}//end class MusicComposer10.java
```

Listing 8 also signals the end of the class named MusicComposer10.

The class named AudioGraph01

A general purpose AudioGraph generator program

AudioGraph01 is a general purpose AudioGraph program that reads an input text file containing numeric values for y as a function of equally spaced values for x and produces an output melody that represents a graph of that data. The values for y are read as a comma-delimited list of values and are treated as type **double** . The name of the text file is input as a command-line parameter.

The input text file

The text file must be stored in a subfolder named **Data** that is a child of the folder containing the compiled program. The text file may be created manually using a simple text editor or it may be created as the output of a program that evaluates a function.

Space characters are not allowed in the data.

The program ignores:

- Blank lines that result in a string with a zero length
- Comment lines that begin with a /
- Lines that begin with a space

Conversion to audio

The data values are converted to audio frequencies and presented as an AudioGraph. The data is biased and scaled so as to make maximum use of the audio dynamic range from 220 Hz to 1760 Hz inclusive.

If the data contains both positive and negative values, the data is adjusted so that the most negative value is emitted at 220 Hz and the most positive value is emitted at 1760 Hz. The frequency that represents zero will fall somewhere between those extremes.

A unique sound is heard whenever a value of zero occurs in the data. It consists of a weighted sum of three frequencies one octave apart centered on the frequency that represents zero.

Three synthetic data items with a value of zero are prepended onto the beginning of the incoming data. They are used to establish the audio pitch for a value of zero on playback.

If the data is all positive, it is biased and scaled so that the minimum value is emitted at 220 Hz and the maximum value is emitted at 1760 Hz. In this case, the frequency that represents zero has little meaning because it is off the bottom of the page, so to speak. It is set at 220 Hz.

An output pulse is heard for each data value. The frequency of the pulse is proportional to the data value. Higher data values result in pulses with a higher pitch. Lower data values result in pulses with a lower pitch.

The output pulse rate in pulses per second is specified by the user as a command-line parameter. Faster output rates provide a quick look at the data. Slower output rates allow for more detailed audio analysis of the data.

To eliminate the pops and clicks that result from abrupt changes in frequency from one pulse to the next, each pulse is shaped using a linear scale factor that is zero at both ends of the pulse and maximum at the center of the pulse.

Sound progresses from the left speaker to the right speaker in proportion to the value of x as a percentage of the total number of x values.

The number, the type, and the order of command-line parameters are defined in the comments in the class named **MusicComposer10** .

Beginning of the class named AudioGraph01

The class named **AudioGraph01** begins in Listing 9 , which includes the declaration and initialization of some instance variables. Listing 9 also includes the beginning of the constructor for the class.

Listing 9 . Beginning of the class named AudioGraph01.

Listing 9 . Beginning of the class named AudioGraph01.

```
import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraph01 extends AudioSignalGenerator02{

  double[] inputData;
  double highFreq = 1760;
  double lowFreq = 220;

  public AudioGraph01(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
       super(audioParams,args,melody);
```

Read the input text file

Listing 10 reads the input text file to define the function that is to be played or filed as an audio file. The text file must be stored in the subfolder named **Data** , which is a child of the folder containing the compiled program. The incoming data is initially stored in an **ArrayList** object for convenience and later transferred to an array object.

The embedded comments in Listing 10 should be sufficient to explain the code.

Listing 10 . Read the input text file.

Listing 10 . Read the input text file.

```
    String fileName = args[2];
    try{
      ArrayList dataList = new ArrayList();
      BufferedReader in = new BufferedReader(
                                          new FileReader("Data/" +
fileName));
      String str;
      while ((str = in.readLine()) != null) {

        //Split the input string into multiple substrings using the
comma as
        // the delimiter and save them in an array object.
        String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() == 0){
          System.out.println("Blank line will be ignored");
        }else if(strArr[0].substring(0,1).equals("/")){
          System.out.println("Comment:  " + Arrays.toString(strArr));
        }else if(strArr[0].substring(0,1).equals(" ")){
          System.out.println("Ignore line that begins with space: " +

Arrays.toString(strArr));
        }else{
          //Apparently good data. Add the contents of the array to the
end of
          // the ArrayList object, one substring (element) at a time.
          for(int cnt = 0;cnt < strArr.length;cnt++){
            dataList.add(Double.parseDouble(strArr[cnt]));
          }//end for loop
        }//end else
      }//end while
      in.close();//close the input file
```

The remainder of the constructor for the class named AudioGraph01

The remainder of the constructor for the class named **AudioGraph01** is shown in Listing 11 . Once again, the embedded comments should be sufficient to explain the code.

Listing 11 . The remainder of the constructor for the class named AudioGraph01.

Listing 11 . The remainder of the constructor for the class named AudioGraph01.

```
      //Move the contents of dataList from the ArrayList object to the
      // inputData array. Make the array three elements longer than the
size
      // of the ArrayList object to accommodate prepending three
elements with
      // a value of zero.
      inputData = new double[dataList.size()+3];
      int count = 0;

      //Prepend three data items with a value of zero to the array. They
will
      // be used to establish the audio pitch for a value of zero.
      inputData[count++] = 0;
      inputData[count++] = 0;
      inputData[count++] = 0;

      Iterator iter = dataList.iterator();
      while(iter.hasNext()){
        double value = (double)iter.next();
        inputData[count++] = value;
      }//end while loop

    }catch(Exception ex){
      ex.printStackTrace();
    }//end catch
  }//end constructor
```

Beginning of the method named getMelody

Listing 8 instantiates a new object of the class named **AudioGraph01** and then calls the **getMelody** method on a reference to that object.

The **getMelody** method reads an array containing data values and returns an array of audio data that relates the data values to frequency or pitch. One output pulse at the correct pitch is produced for each data value. Out pulses representing a data value of zero are constructed from three frequencies to make them sound different.

The output rate in pulses per second is provided by the user as a command-line parameter.

The audio output can be thought of as an audio representation of a graph of the input data with three zero values prepended onto the front to establish the pitch for a y value of zero. Other than those three pulses, each output pulse represents one point on the graph. That is, each output pulse represents the value of y for a given value of x for the function being evaluated.

The beginning of the method named **getMelody** is shown in Listing 12 .

You have seen code like this in many previous modules, so no explanation beyond the embedded comments should be needed for this code.

Listing 12 . Beginning of the method named getMelody.

```
  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample
rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    //Declare variables used to control the output volume on the left
and
    // right speaker channels. These values will be used to cause pulses
    // representing the data values to progress uniformly from the left
    // speaker to the right speaker in proportion to the value of x.
    double gain = 0.0;
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare a variable that is used to control the frequency of each
pulse.
    double freq = 0.0;

    //Set the length of each pulse in seconds and in samples. The user
    // specifies the output rate in pulses per second as a command-line
    // parameter. The pulse length in seconds is the reciprocal of that
value.
    double pulseLengthInSec = 1/Double.parseDouble(args[1]);//in seconds
    int pulseLengthInSamples = (int)
(pulseLengthInSec*audioParams.sampleRate);

    //Create an output array of sufficient size to contain the audio
data.
    melody = new byte[(int)(inputData.length *
                            pulseLengthInSamples *
                            bytesPerSampPerChan *
                            audioParams.channels)];
    System.out.println("melody.length = " + melody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);
```

Determine the frequency and data ranges

Later on, in order to map the data values into frequencies and make maximum use of the available audio dynamic range from 220 Hz to 1760 Hz, we will need to know the frequency range and the data range.

The first statement in Listing 13 computes the frequency range. I could have taken care of this back in Listing 9 where the high and low frequency limits were established but I overlooked it at that point.

The code beginning with the second statement determines the minimum and maximum data values. This can only be determined after the actual data values have been read from the input file. These values will be used to bias and scale the data so as to make maximum use of the available audio dynamic range.

Listing 13 . Determine the frequency and data ranges.

```
double freqRange = highFreq - lowFreq;

double highData = Double.MIN_VALUE;
double lowData = Double.MAX_VALUE;
//Skip first three values which are always zero.
for(int cnt = 3;cnt < inputData.length;cnt++){
  if (inputData[cnt] > highData){
    highData = inputData[cnt];
  }//end if
  if(inputData[cnt] < lowData){
    lowData = inputData[cnt];
  }//end if
}//end for loop

double dataRange = highData - lowData;
```

Determine the frequency to represent a y-value of zero

Listing 14 determines the frequency that will represent a y-value of 0. If the data are all positive, the data values will be biased so that the lowest value will sound at 220 Hz and the highest value will sound at 1760 Hz. Beyond that, the code in Listing 14 shouldn't require an explanation.

Listing 14 . Determine the frequency to represent a y-value of zero.

Listing 14 . Determine the frequency to represent a y-value of zero.

```
    double zeroFreq;
    if(lowData >= 0.0){
      //Bias all data so that the lowest value will display at 220Hz.
Skip the
      // first three data values which are always zero.
      for(int cnt = 3;cnt < inputData.length;cnt++){
        inputData[cnt] -= lowData;
      }//end for loop

      //zeroFreq has little meaning in this case because it is probably
off the
      // bottom of the page. Set it to the bottom of the range.
      zeroFreq = lowFreq;
    }else{
      //Set zeroFreq to a value that represents a value of zero for
bipolar
      // data.
      zeroFreq = lowFreq +(Math.abs(lowData)/dataRange) * freqRange;
    }//end else
    System.out.println("zeroFreq = " + zeroFreq);
```

Compute audio sample values and deposit them in the output array

Listing 15 computes audio sample values and deposits them in the output array.

You have seen code similar to this in previous modules. The only thing that is new and different here is the algebra that is used to map the sample values to frequencies and the algebra that is used to cause the sound to progress from the left speaker to the right speaker in proportion to the value of x currently being evaluated.

Note that the sound that is produced for a y-value of zero is different from the sound that is produced for all other values. In the case of a y-value of zero, the sound that is produced is the weighted sum of three sounds covering two octaves centered on the frequency that is considered to be the zero frequency. This makes it possible to audibly identify those points where the function being graphed crosses the horizontal or zero axis *(provided that there is actually a data value of zero)* .

Beyond that, the embedded comments should be sufficient to explain the code in Listing 15 .

Listing 15 . Compute audio sample values and deposit them in the output array.

```
     int sampLength =
melody.length/audioParams.channels/bytesPerSampPerChan;

    for(int cnt = 0; cnt < sampLength; cnt++){
```

Listing 15 . Compute audio sample values and deposit them in the output array.

```
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      double yValue = 0;
      if(cnt%pulseLengthInSamples == 0){
        //It is time for a new pulse. Get the next y value from the data
array
        // and use it to compute the frequency of the next pulse.
        yValue = inputData[cnt/pulseLengthInSamples];
        freq = zeroFreq + (yValue/dataRange)*freqRange;
      }//end if

      //Deposit audio data in the melody array for each channel. Shape
the
      // amplitude of each pulse with a triangular scale factor (rooftop
shape)
      // to  minimize the undesirable pops and clicks that occur when
there
      // are abrupt change in the frequency from one pulse to the next.
The
      // following gain factor ranges from 0.0 at the ends to maximum in
the
      // center of the pulse.
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -

cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;
      }//end if

      //Set the final gain to a value that is compatible with 16-bit
audio
      // data.
      gain = 8000*gain;

      //Cause the sound to progress from the left speaker to the right
speaker
      // in proportion to the value of x.
      rightGain = gain * ((double)cnt/sampLength);
      leftGain = gain - rightGain;

      if(freq == zeroFreq){
        //Compute scaled pulse values and deposit them into the melody.
Mark
        // the zeroFreq by adding frequency components at a reduced
level one
        // octave above and one octave below the zeroFreq. This will
make it
        // sound special in the output.
        byteBuffer.putShort((short)
(leftGain*Math.sin(2*Math.PI*freq*time) +

(leftGain*Math.sin(2*2*Math.PI*freq*time))/3 +
```

Listing 15 . Compute audio sample values and deposit them in the output array.

```
(leftGain*Math.sin(2*Math.PI*freq*time/2))/3));
        byteBuffer.putShort((short)
(rightGain*Math.sin(2*Math.PI*freq*time) +

(rightGain*Math.sin(2*2*Math.PI*freq*time))/3 +

(rightGain*Math.sin(2*Math.PI*freq*time/2))/3));
      }else{
        //Compute scaled pulse values and deposit them into the melody.
        byteBuffer.putShort((short)
(leftGain*Math.sin(2*Math.PI*freq*time)));
        byteBuffer.putShort((short)
(rightGain*Math.sin(2*Math.PI*freq*time)));
      }//end else
    }//end for loop

    return melody;
  }//end method getMelody
  //--------------------------------------------------------------------
-----//

}//end class AudioGraph01
```

Listing 15 also signals the end of the class named **AudioGraph01** .

Run the program

I encourage you to copy the code from Listing 16 through Listing 20 *(or use the code that is provided in the downloadable zip file)* . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

As mentioned earlier, assuming that you have the Java Development Kit installed on your computer, the zip file that you can download contains all of the files necessary to replicate the audio output for all of the functions illustrated by the audio files in the above list . Simply extract the contents of the zip file into an empty folder, execute the **...Solver.bat** files to create the data for the functions, and execute the **...Player.bat** files to hear the audio for the functions and to write the audio into output audio files of type AU.

When you compile this program, you will probably see the following warnings:

> Note: .\PlayerPiano01.java uses unchecked or unsafe operations.

> Note: Recompile with -Xlint:unchecked for details.

You can simply ignore those warnings if you choose to do so. Otherwise, you can learn how to eliminate the warnings by doing the following:

- Go to the book Object-Oriented Programming (OOP) with Java -> Contents -> ITSE2317 -> Generics -> Java4210: Getting Started with Generics
- Study that module and the seven modules following it as listed in the Contents.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jbs2070-A General Purpose AudioGraph Program
- File: Jbs2070.htm
- Published: 09/01/14
- Revised: 09/29/15

Complete program listings

Complete program listings are provided below. Source code for these listings as well as other source code discussed in this module is available in a downloadable zip file .

Listing 16 . The program named Sinc01.

```
/*File Sinc01.java
Copyright 2014, R.G.Baldwin
Revised 08/31/14

This program produces an output text file containing values for y as a funct
of x. The contents of the text file can be converted to audio and either pla
```

Listing 16 . The program named Sinc01.

```
or written into an audio file of type AU for playback later by the program
named AudioGraph01.


Function:
y = Math.sin(2*Math.PI*f*x)/x
This is the classic sin(x)/x function, otherwise known a the sinc
function. It appears frequently in digital signal processing (DSP).

To modify this program to handle other functions, you only need to modify th
instance variables in the class named Runner and modify the code in the meth
named getYval.

Tested using Java SE 8 under Win 7
**********************************************************************/


import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class Sinc01{//Driver class
  public static void main(String[] args){
    //Do not modify the code in this method.
    Runner obj = new Runner();
    obj.run();
  }//end main
}//end class Sinc01
//===================================================================

class Runner{
  //Modify the following instance variables as needed.
  double xMin = -20;//Minimum value for x
  double xMax = 20;//Maximum value for x
  double xInc = 0.25;//Used to determine x-values for evaluation of y-value

  String fileName = "Data/Sinc01.txt";//Output file name in Data folder

  void run(){
    //Do not modify the code in this method.
    double xVal = xMin;
    String outString = "";
    while(xVal <= xMax){
      //Construct an output string contain comma-separated values of y as a
      // function of x.
      outString += getYval(xVal) + ",";
      //Increment x
      xVal += xInc;
    }//end while loop

    writeOutputFile(fileName,outString);

  }//end run method
  //-------------------------------------------------------------------
```

Listing 16 . The program named Sinc01.

```
  //This method evaluates the function. Modify it to evaluate different
  // functions.
  //Function:
  //y = Math.sin(2*Math.PI*f*x)/x
  //This is the classic sin(x)/x function, otherwise known a the sinc
  // function. It appears frequently in digital signal processing (DSP).
  double getYval(double xVal){
    //Evaluate the function here
    double f = 0.25;
    double result = 0;
    if(xVal != 0.0){
      result = Math.sin(2*Math.PI*f*xVal)/xVal;
    }else{
      //Don't divide by zero
      result = Math.sin(2*Math.PI*f*0.00001)/0.00001;
    }//end else

    //Limit the return value to three decimal digits
    return (Math.rint(1000.0*result))/1000.0;
  }//end getY
  //---------------------------------------------------------------------

  //This method writes the output file. Do not modify this method.
  void writeOutputFile(String fileName,String outString){
    try{
      File file = new File(fileName);
       //if the file doesn't exists, create it. If it does exist, overwrite
      if (!file.exists()) {
        file.createNewFile();
      }//end if
      FileWriter fileWriter = new FileWriter(file.getAbsoluteFile());
      BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
      bufferedWriter.write(outString);
      bufferedWriter.close();
    }catch (IOException e) {
      e.printStackTrace();
    }//end catch
  }//end writeOutputFile
  //---------------------------------------------------------------------
}//end class Runner
```

...

Listing 17 . The file named Sinc01Solver.bat.

Listing 17 . The file named Sinc01Solver.bat.

```
echo off
del *.class

javac Sinc01.java
java Sinc01

del *.class

pause
```

...

Listing 18 . The file named Sinc01Player.bat

```
echo off
del *.class

del Sinc01.au

echo on
javac MusicComposer10.java

java MusicComposer10 play 3 Sinc01.txt
java MusicComposer10 Sinc01 3 Sinc01.txt

echo off
del *.class

pause
```

...

Listing 19 . The class named MusicComposer10.

```
/*File MusicComposer10.java
```

Listing 19 . The class named MusicComposer10.

```
Copyright 2014, R.G.Baldwin
Revised 08/23/14

This is a general purpose AudioGraph program that reads an input text file
containing numeric values for y as a function of equally spaced values for x
and produces an output melody that represents a graph of that data. The valu
for y are read as a comma-delimited list of values and are treated as type
double. The name of the text file is input as a command-line parameter.

Additional details will be found in the class named AudioGraph01.

This program requires the following classes:

AudioGraph01
AudioSignalGenerator02
AudioPlayOrFile01
AudioFormatParameters01

The sound can be played immediately or can be saved in an audio file of
type AU for playback later. You should be able to play the audio file with a
standard media player that can handle the AU file type

Tested using JDK 1.8 under Win 7.
**************************************************************************/

public class MusicComposer10{
  //Instantiate an object containing audio format parameters with predefined
  // values. They may be modified by the signal generator at runtime. Values
  // allowed by Java SDK 1.4.1 are shown in comments in the class definition
  AudioFormatParameters01 audioParams = new AudioFormatParameters01();

  //A buffer to hold the audio data that will be played or filed.
  byte[] melody;

  //A place to store the incoming args array.
  String[] args;
  //--------------------------------------------------------------------//

  //Command-line parameter (three parameters required)
  //0 - If "play", the sound will be played immediately. Otherwise, the stri
  //    will be used as a filename for an audio file of type AU. In the latt
  //    case, it must be a string that would be valid as a file name for the
  //    operating system in use.
  //1 - Output pulses per second
  //2 - Input file name
  public static void main(String[] args){
    //Instantiate a new object of this class.
    new MusicComposer10(args);
  }//end main
  //--------------------------------------------------------------------//

  public MusicComposer10(String[] args){//constructor
    //Save the args array.
    this.args = args;
```

Listing 19 . The class named MusicComposer10.

```
    //Create default args data if no args data is provided on the command li
    if(args.length == 0){
      this.args = new String[3];
      this.args[0] = "play";//Play the melody immediately
      this.args[1] = "6";//Pulses per minute
      this.args[2] = "TestData01.txt";
    }//end if

    //Get a populated array containing audio data.
    AudioGraph01 audioGraph01 = new AudioGraph01(audioParams,this.args,meloc
    melody = audioGraph01.getMelody();

    //Play or file the audio data
    new AudioPlayOrFile01(audioParams,melody,this.args[0]).playOrFileData();
  }//end constructor
  //-----------------------------------------------------------------------
}//end class MusicComposer10.java
//=======================================================================
```

...

Listing 20 . The class named AudioGraph01.

```
/*File AudioGraph01.java
Copyright 2014, R.G.Baldwin
Revised 08/31/14

This is a general purpose AudioGraph program that reads an input text file
containing numeric values for y as a function of equally spaced values for x
and produces an output melody that represents a graph of that data. The valu
for y are read as a comma-delimited list of values and are treated as type
double. The name of the text file is input as a command-line parameter.

The text file must be stored in a subfolder named Data that is a child of th
folder containing the compiled program. The text file may be created manuall
using a simple text editor or may be created as the output of a program that
evaluates a function. Space characters are not allowed in the data.

The program ignores:
 Blank lines that result in a string with a zero length
 Comment lines that begin with a /
 Lines that begin with a space

The data is biased and scaled so as to make maximum use of the audio dynamic
```

Listing 20 . The class named AudioGraph01.

```
range from 220 Hz to 1760 Hz. If the data contains both positive and negativ
values, the data is adjusted so that the most negative value is emitted at
220 Hz and the most positive value is emitted at 1760 Hz. The frequency that
represents zero will fall somewhere between those extremes. A unique sound
is heard whenever a value of zero occurs in the data. It consists of a weigh
sum of three frequencies one octave apart centered on the frequency
that represents zero.

Three data items with a value of zero are prepended onto the incoming data.
They are used to establish the audio pitch for a value of zero.

If the data is all positive, it is biased and scaled so that the minimum val
is emitted at 220 Hz and the maximum value is emitted at 1760 Hz. In this ca
the frequency that represents zero has little meaning because it is off the
bottom of the page, so to speak. It is set at 220 Hz.

An output pulse is heard for each data value. The frequency of the pulse is
proportional to the data value. Higher data values result in pulses with a
higher pitch. Lower data values result in pulses with a lower pitch.

The output pulse rate in pulses per second is specified by the user as a
command-line parameter. Faster output rates provide a quick look at the data
Slower output rates allow for more detailed audio analysis of the data.

To eliminate the pops and clicks that result from abrupt changes in frequenc
from one pulse to the next, each pulse is shaped using a linear scale factor
that is zero at both ends of the pulse and maximum at the center of the puls

Sound progresses from the left speaker to the right speaker in proportion to
the value of x as a percentage of the total number of x values.

The number, type, and order of command-line parameters are defined in the
comments in the class named MusicComposer10.

Tested using Java SE with Windows 7.
**************************************************************************

import java.io.*;
import java.nio.*;
import java.util.*;

public class AudioGraph01 extends AudioSignalGenerator02{

  double[] inputData;
  double highFreq = 1760;
  double lowFreq = 220;

  public AudioGraph01(AudioFormatParameters01 audioParams,
              String[] args,
              byte[] melody){
    super(audioParams,args,melody);


    //Read the input file to define the function that is to be played or
```

Listing 20 . The class named AudioGraph01.

```
    // filed. It must be stored in the subfolder named Data. The data is
    // initially stored in an ArrayList object for convenience and later
    // transferred to an array object.
    String fileName = args[2];
    try{
      ArrayList dataList = new ArrayList();
      BufferedReader in = new BufferedReader(
                                          new FileReader("Data/" + fileName
      String str;
      while ((str = in.readLine()) != null) {

        //Split the input string into multiple substrings using the comma as
        // the delimiter and save them in an array object.
        String[] strArr = str.split(",");

        //Ignore:
        // Blank lines that result in a string with a zero length
        // Comments that begin with a /
        // Lines that begin with a space
        if(strArr[0].length() == 0){
          System.out.println("Blank line will be ignored");
        }else if(strArr[0].substring(0,1).equals("/")){
          System.out.println("Comment:  " + Arrays.toString(strArr));
        }else if(strArr[0].substring(0,1).equals(" ")){
          System.out.println("Ignore line that begins with space: " +
                                                  Arrays.toString(strArr
        }else{
          //Apparently good data. Add the contents of the array to the end o
          // the ArrayList object, one substring (element) at a time.
          for(int cnt = 0;cnt < strArr.length;cnt++){
            dataList.add(Double.parseDouble(strArr[cnt]));
          }//end for loop
        }//end else
      }//end while
      in.close();//close the input file

      //Move the contents of dataList from the ArrayList object to the
      // inputData array. Make the array three elements longer than the size
      // of the ArrayList object to accommodate prepending three elements wi
      // a value of zero.
      inputData = new double[dataList.size()+3];
      int count = 0;

      //Prepend three data items with a value of zero to the array. They wil
      // be used to establish the audio pitch for a value of zero.
      inputData[count++] = 0;
      inputData[count++] = 0;
      inputData[count++] = 0;

      Iterator iter = dataList.iterator();
      while(iter.hasNext()){
        double value = (double)iter.next();
        inputData[count++] = value;
      }//end while loop
```

Listing 20 . The class named AudioGraph01.

```
    }catch(Exception ex){
      ex.printStackTrace();
    }//end catch
  }//end constructor
  //--------------------------------------------------------------------

  //This method reads an array containing data values and produces an array
  // output audio data that relates the data values to frequency or pitch.
  // One output pulse is produced for each data value. The output rate in
  // pulses per second is provided by the user as a command-line parameter.
  //The audio output can be thought of as an audio representation of a graph
  // of the input data with three zero values prepended onto the front to
  // establish the pitch for a y value of zero.

  byte[] getMelody(){
    //Set channels to 2 for stereo overriding the default value of 1.
    audioParams.channels = 2;

    //Each channel requires two 8-bit bytes per 16-bit sample.
    int bytesPerSampPerChan = 2;

    //Override the default sampleRate of 16000.0F. Allowable sample rates
    // are 8000,11025,16000,22050, and 44100 samples per second.
    audioParams.sampleRate = 8000.0F;

    //Declare variables used to control the output volume on the left and
    // right speaker channels. These values will be used to cause pulses
    // representing the data values to progress uniformly from the left
    // speaker to the right speaker in proportion to the value of x.
    double gain = 0.0;
    double leftGain = 0.0;
    double rightGain = 0.0;

    //Declare a variable that is used to control the frequency of each pulse
    double freq = 0.0;

    //Set the length of each pulse in seconds and in samples. The user
    // specifies the output rate in pulses per second as a command-line
    // parameter. The pulse length in seconds is the reciprocal of that valu
    double pulseLengthInSec = 1/Double.parseDouble(args[1]);//in seconds
    int pulseLengthInSamples = (int)(pulseLengthInSec*audioParams.sampleRate

    //Create an output array of sufficient size to contain the audio data.
    melody = new byte[(int)(inputData.length *
                            pulseLengthInSamples *
                            bytesPerSampPerChan *
                            audioParams.channels)];
    System.out.println("melody.length = " + melody.length);

    //Prepare a ByteBuffer for use
    byteBuffer = ByteBuffer.wrap(melody);

    double freqRange = highFreq - lowFreq;
```

Listing 20 . The class named AudioGraph01.

```
    //Determine the minimum and maximum data values. These values will be us
    // to bias and scale the data so as to make maximum use of the available
    // audio dynamic range from 220 Hz to 1760 Hz.
    double highData = Double.MIN_VALUE;
    double lowData = Double.MAX_VALUE;
    //Skip first three values which are always zero.
    for(int cnt = 3;cnt < inputData.length;cnt++){
      if (inputData[cnt] > highData){
        highData = inputData[cnt];
      }//end if
      if(inputData[cnt] < lowData){
        lowData = inputData[cnt];
      }//end if
    }//end for loop

    double dataRange = highData - lowData;

    //Determine the frequency that will represent a y-value of 0. Also, if t
    // data is all positive, bias the data values so that the lowest value
    // will sound at 220Hz and the highest value will sound at 1760 Hz.
    double zeroFreq;
    if(lowData >= 0.0){
      //Bias all data so that the lowest value will display at 220Hz. Skip t
      // first three data values which are always zero.
      for(int cnt = 3;cnt < inputData.length;cnt++){
        inputData[cnt] -= lowData;
      }//end for loop

      //zeroFreq has little meaning in this case because it is probably off
      // bottom of the page. Set it to the bottom of the range.
      zeroFreq = lowFreq;
    }else{
      //Set zeroFreq to a value that represents a value of zero for bipolar
      // data.
      zeroFreq = lowFreq +(Math.abs(lowData)/dataRange) * freqRange;
    }//end else
    System.out.println("zeroFreq = " + zeroFreq);

    //Compute the audio sample values and deposit them in the output melody
    // array.
    int sampLength = melody.length/audioParams.channels/bytesPerSampPerChan;

    for(int cnt = 0; cnt < sampLength; cnt++){
      //Compute the time in seconds for this sample.
      double time = cnt/audioParams.sampleRate;

      double yValue = 0;
      if(cnt%pulseLengthInSamples == 0){
        //It is time for a new pulse. Get the next y value from the data arr
        // and use it to compute the frequency of the next pulse.
        yValue = inputData[cnt/pulseLengthInSamples];
        freq = zeroFreq + (yValue/dataRange)*freqRange;
      }//end if
```

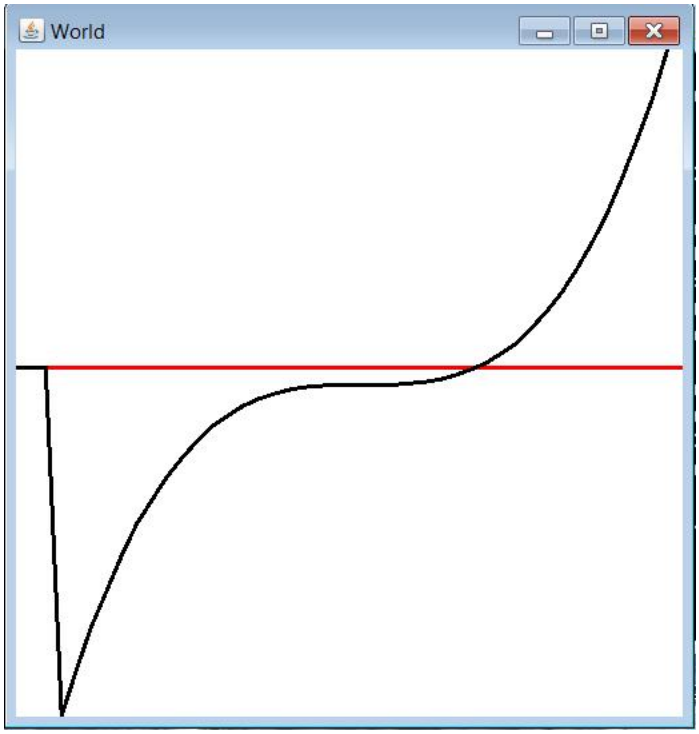
Listing 20 . The class named AudioGraph01.

```
      //Deposit audio data in the melody array for each channel. Shape the
      // amplitude of each pulse with a triangular scale factor (rooftop sha
      // to  minimize the undesirable pops and clicks that occur when there
      // are abrupt change in the frequency from one pulse to the next. The
      // following gain factor ranges from 0.0 at the ends to maximum in the
      // center of the pulse.
      gain = (cnt%pulseLengthInSamples)/(double)pulseLengthInSamples;

      if(gain > 0.5){
        //Change to a negative slope.
        gain = (pulseLengthInSamples -
                        cnt%pulseLengthInSamples)/(double)pulseLengthInSampl
      }//end if

      //Set the final gain to a value that is compatible with 16-bit audio
      // data.
      gain = 8000*gain;

      //Cause the sound to progress from the left speaker to the right speak
      // in proportion to the value of x.
      rightGain = gain * ((double)cnt/sampLength);
      leftGain = gain - rightGain;

      if(freq == zeroFreq){
        //Compute scaled pulse values and deposit them into the melody. Mark
        // the zeroFreq by adding frequency components at a reduced level or
        // octave above and one octave below the zeroFreq. This will make it
        // sound special in the output.
        byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time) +
                              (leftGain*Math.sin(2*2*Math.PI*freq*time))/3 +
                              (leftGain*Math.sin(2*Math.PI*freq*time/2))/3))
        byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)
                              (rightGain*Math.sin(2*2*Math.PI*freq*time))/3
                              (rightGain*Math.sin(2*Math.PI*freq*time/2))/3)
      }else{
        //Compute scaled pulse values and deposit them into the melody.
        byteBuffer.putShort((short)(leftGain*Math.sin(2*Math.PI*freq*time)))
        byteBuffer.putShort((short)(rightGain*Math.sin(2*Math.PI*freq*time)))
      }//end else
    }//end for loop

    return melody;
  }//end method getMelody
  //----------------------------------------------------------------------

}//end class AudioGraph01
//======================================================================
```

Figures

The following images are provided for the benefit of those students who have the ability to create embossed images. There is one image for each AudioGraph listed above . These images differ from the AudioGraph version only in that a horizontal axis is drawn on the image while a special sound marks zero crossings in the AudioGraph. The downloadable zip file contains the JPEG files for these images.

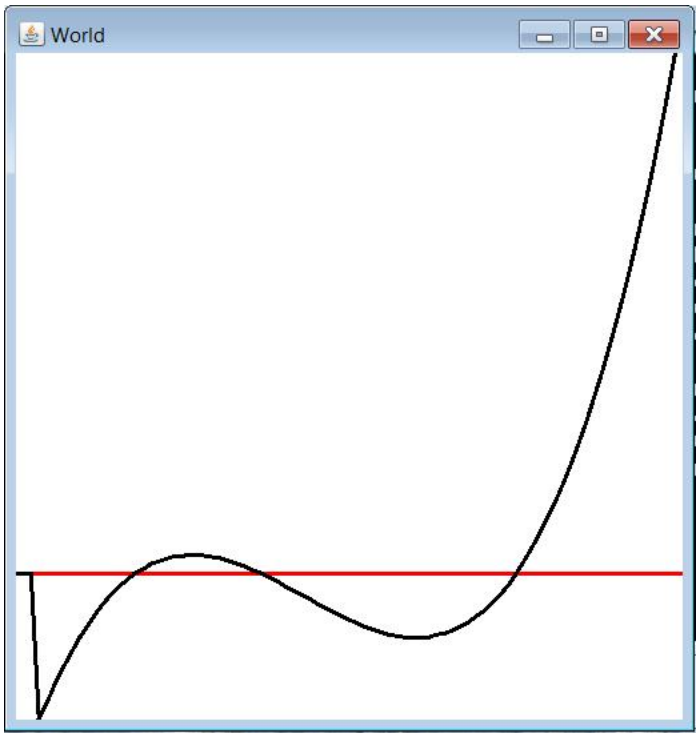
Figure 1 . Cubic01.



...

Figure 2 . Cubic02.

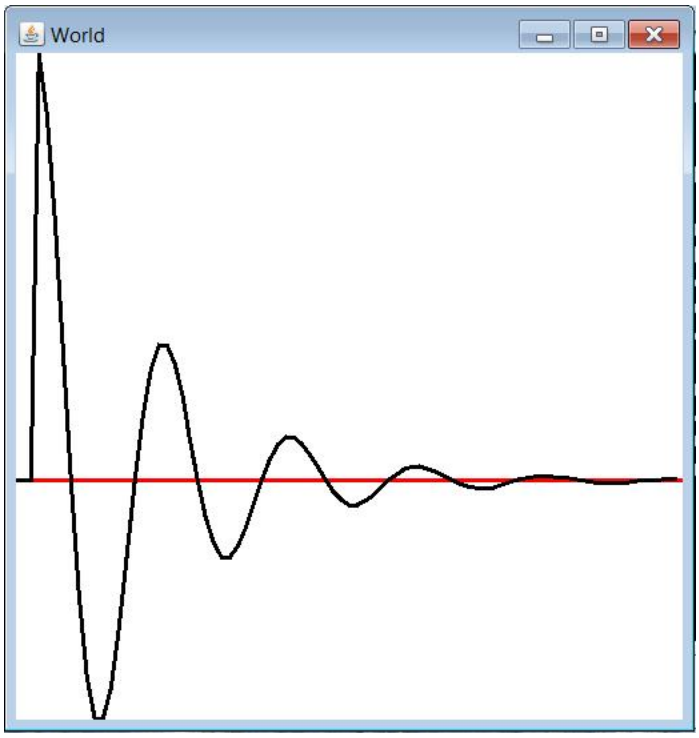
Figure 2 . Cubic02.



...

Figure 3 . DampedSinusoid01.

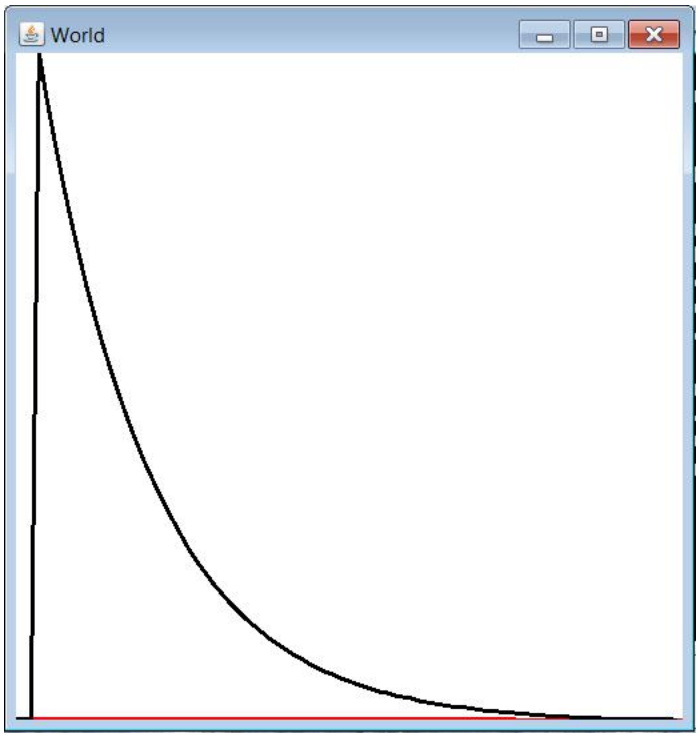
Figure 3 . DampedSinusoid01.



...

Figure 4 . ExponentialDamper01.

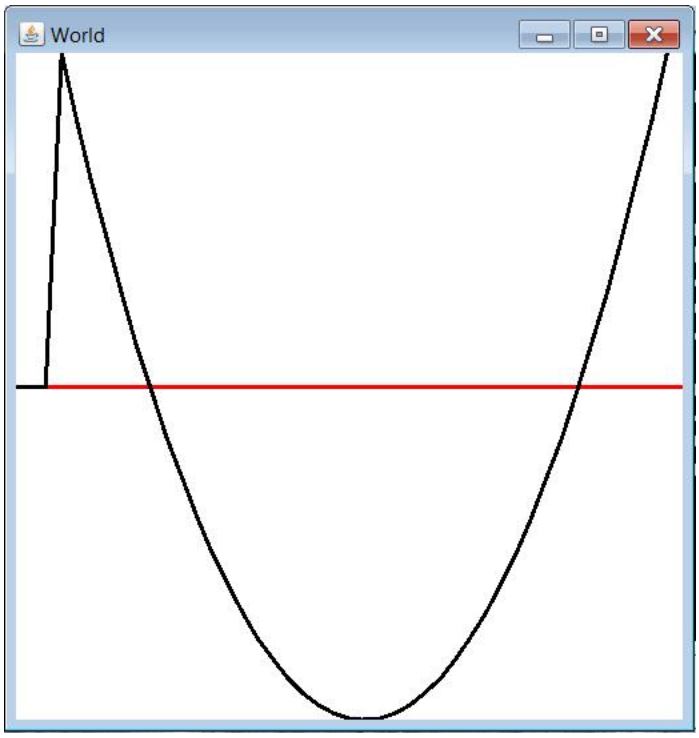
Figure 4 . ExponentialDamper01.



...

Figure 5 . Parabola01.

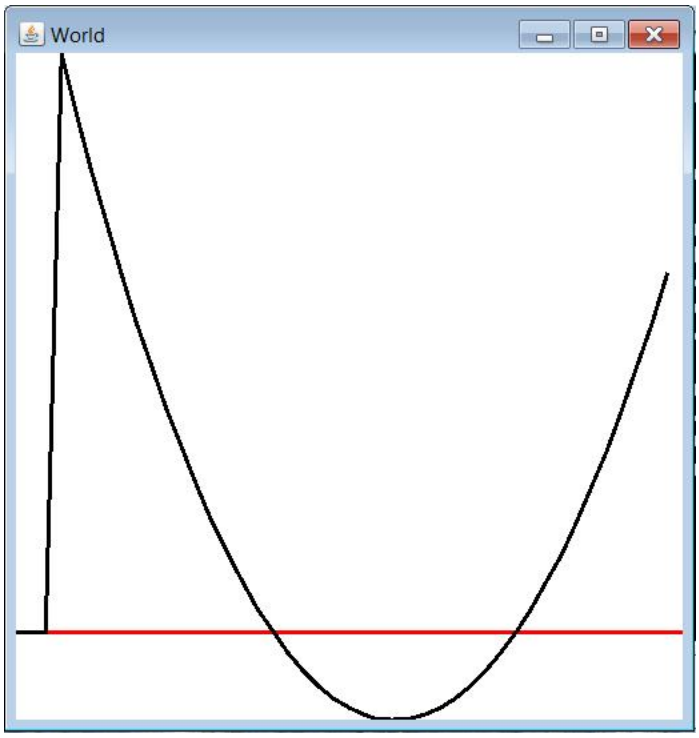
Figure 5 . Parabola01.



...

Figure 6 . Quadratic01.

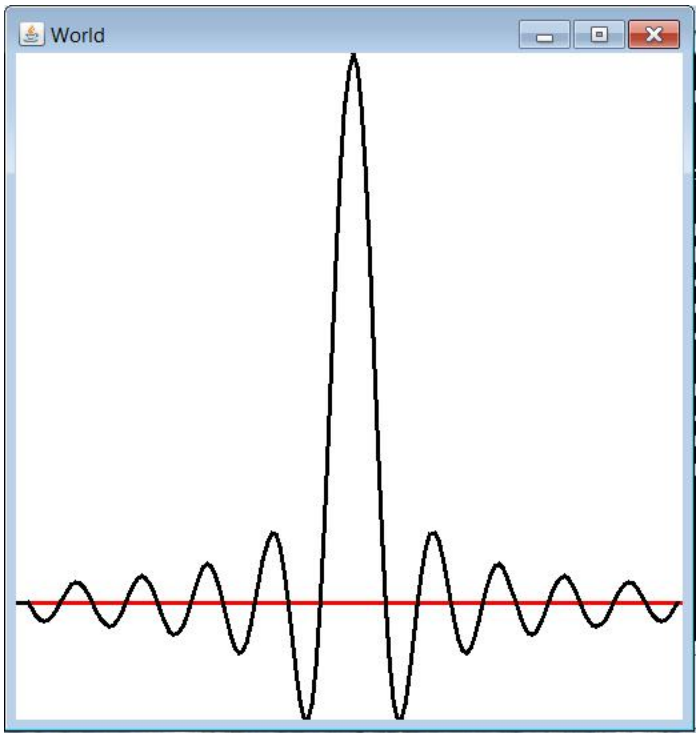
Figure 6 . Quadratic01.



...

Figure 7 . Sinc01.

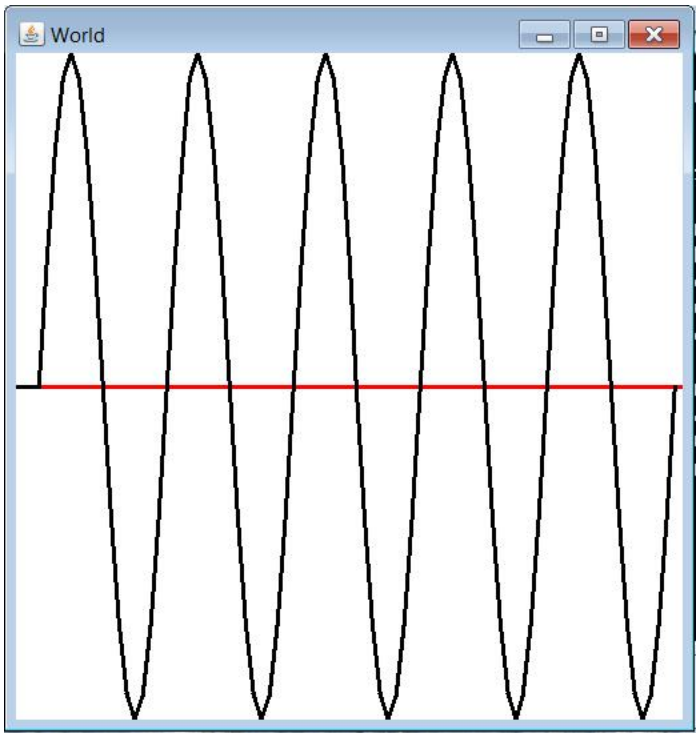
Figure 7 . Sinc01.



...

Figure 8 . Sinusoid01.

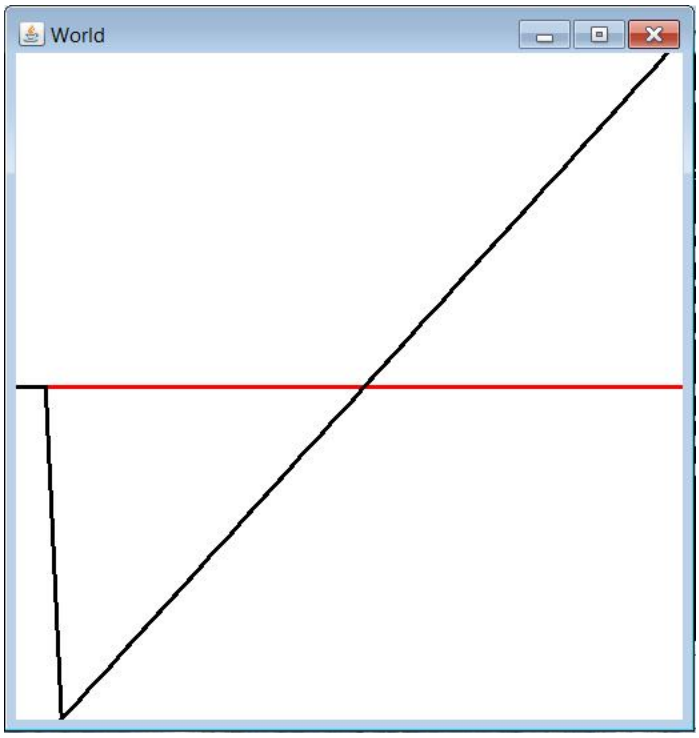
Figure 8 . Sinusoid01.



...

Figure 9 . StraightLine01

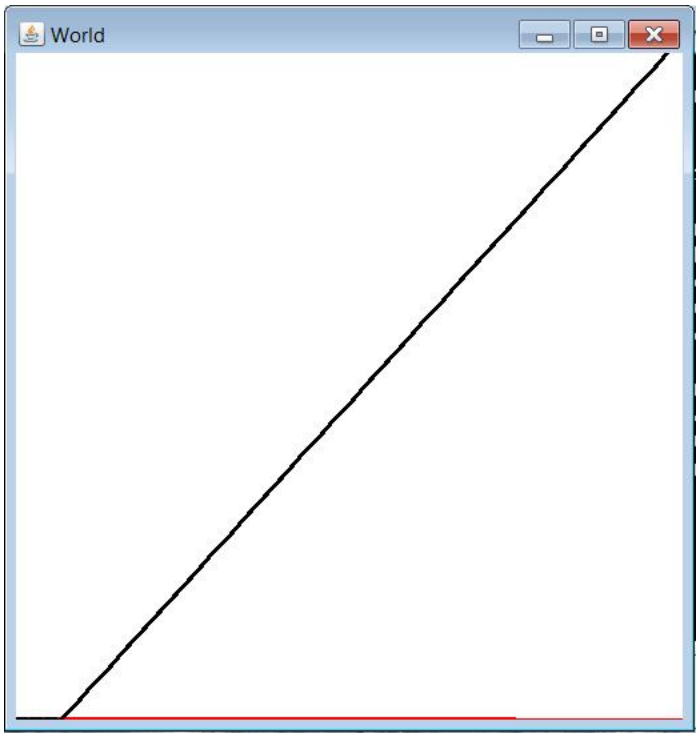
Figure 9 . StraightLine01



...

Figure 10 . StraightLine02.

Figure 10 . StraightLine02.



-end-